Edition Notice

Note

Before using this information and the product it supports, be sure to read the general information under Notices.

Fourth Edition (February 1999)

This edition applies to the IBM TCP/IP Version 4.21 for OS/2 Warp licensed program.

In the U.S., customers can order publications by calling IBM Software Manufacturing Solutions at 1 800-879-2755. Outside the U.S., customers should contact the IBM branch office serving their locality.

About This Information

This *IBM TCP/IP for OS/2 Warp Programming Reference* describes the routines for application programming in the TCP/IP for OS/2 Warp environment on a workstation.

This edition applies to the IBM TCP/IP Version 4.21 for OS/2 Warp licensed program.

This section describes:

- Primary users of this information
- Content of each major information division
- Conventions used in this information
- New functions in this information
- Where to find more information

Who Should Use This Information

This information is intended for application and system programmers with experience in writing application programs on a workstation. You should also be familiar with the OS/2 operating system and know multitasking operating system concepts. It is important that you also know the C programming language.

If you are not familiar with TCP/IP concepts, see *Internetworking With TCP/IP Volume I: Principles, Protocols, and Architecture* , and *Internetworking With TCP/IP Volume II: Implementation and Internals* .

What This Information Describes

This document contains guidance and reference information about the following topics.

Guidance Information

- Sockets General Programming Information

  Describes the TCP/IP socket interface and how to use the socket routines in a user-written application.

- Sockets in the Internet Domain

Describes TCP/IP network protocols, getting started with sockets in the Internet domain, Internet address formats, and TCP/IP-specific network utility routines.

- Sockets over Local IPC

  Describes how programmers can communicate on the same machine using the sockets API, and the local IPC address format.

- Sockets over NetBIOS

  Describes how programmers can communicate with NetBIOS using the sockets API, and the NetBIOS address format.

- Windows Sockets Version 1.1 for OS/2

  Presents information for implementing the Winsock 1.1 API for OS/2 applications.

- Remote Procedure Calls

  Describes the remote procedure calls and how to use them in a user-written application.

- File Transfer Protocol

  Describes the file transfer protocol routines and how to use them in a user-written application.

- Resource ReSerVation Protocol

  Describes the resource reservation protocol routines and how to use them in a user-written application.

Reference Information

- Protocol-Independent C Sockets API

  Describes the protocol-independent socket calls supported by networking services. This information includes call syntax, usage, and related information.

- TCP/IP Network Utility Routines API

  Describes the sockets utility and Sockets Secure Support (SOCKS) function calls supported by networking services. This information includes call syntax, usage, and related information.

- Remote Procedure and eXternal Data Representation API

  Describes the remote procedure and XDR function calls along with their syntax, usage, and related information.

- File Transfer Protocol API

  Describes the file transfer protocol function calls along with their syntax, usage, and related information.

- Resource ReSerVation Protocol API

  Describes the resource reservation protocol function calls along with their syntax, usage, and related information.

Appendixes

- NETWORKS File Structure

  Provides examples of network names contained in the TCPIP\ETC\NETWORKS file.

- Socket Error Constants

  Provides the socket error codes and descriptions.

- Well-Known Port Assignments

  Provides a list of the well-known ports supported by TCP/IP.

- Notices

  Contains copyright notices, disclaimers, and trademarks relating to TCP/IP for OS/2 Warp.

Conventions Used in This Information

How the Term "Internet" Is Used

An *internet* is a logical collection of networks supported by gateways, routers, bridges, hosts, and various layers of protocols that permit the network to function as a large, virtual network.

The term internet is used as a generic term for a TCP/IP network and should not be confused with the *Internet* (note capital I), which consists of large national backbone networks (such as MILNET, NSFNet, and CREN) and myriad regional and local campus networks all over the world.

What Is New in This Information

For the list of changes that have been made since earlier versions of TCP/IP for OS/2, see

- Changes in Version 4.21.
- Changes in Version 4.21.
- Changes in Version 4 (Merlin).

Changes in Version 4.21

The following new API functions have been added:

1. accept_and_recv()
2. send_file()

Also, a description for new library, R0LIB32, has been added.

Changes in Version 4.21

Descriptions of TCP Extensions for Transactions (T/TCP), TCP Extensions for High Performance, and High Performance Send (HPS) have been added to Sockets General Programming Information.

The Resource ReSerVation Protocol (RSVP) has been added. See:

Resource ReSerVation Protocol
Resource ReSerVation Protocol API

Support for Berkley Software Distribution (BSD) Version 4.4 is added.

In Protocol-Independent C Sockets API, changes have been made to getsockopt() and setsockopt(). sysctl() has been added. The ioctl() call has been split into os2_ioctl() for OS/2 and ioctl() for BSD, and the select() call has been split into os2_select() for OS/2 and select() for BSD. The sock_init() call has been removed.

In TCP/IP Network Utility Routines API, the bswap(), lswap() and tcp_h_errno() calls have been removed.

In File Transfer Protocol API, additions have been made to the *host* parameter to allow you to specify the port number used by the FTP server.

Changes in Version 4 (Merlin)

This edition reorganizes the information from the previous edition into guidance and reference sections, adds substantially to the sockets guidance material in Sockets General Programming Information and Sockets in the Internet Domain, and also adds the following function calls:

In Protocol-Independent C Sockets API:

addsockettolist()
removesocketfromlist()

In TCP/IP Network Utility Routines API:

dn_find()
dn_skipname()
_getlong()
_getshort()
h_errno
putlong()
putshort()
Raccept()
Rbind()
Rconnect()
res_query()
res_querydomain()
res_search()
rexec()
Rgethostbyname()
Rgetsockname()
Rlisten()

In addition, the os2_connect() and os2_gethostbyname() calls have been removed; see connect() and gethostbyname() instead.

For More Information

You may purchase this information in printed form by ordering IBM publication number SC31-8407. U.S. customers can order by calling IBM Software Manufacturing Solutions at 1-800-879-2755. Outside the U.S., customers should contact the IBM branch office serving their locality.

Guidance Information

This section describes:

- Sockets General Programming Information

  Describes the TCP/IP socket interface and how to use the socket routines in a user-written application.

- Sockets in the Internet Domain

  Describes TCP/IP network protocols, getting started with sockets in the Internet domain, Internet address formats, and TCP/IP-specific network utility routines.

- Sockets over Local IPC

  Describes how programmers can communicate on the same machine using the sockets API, and the local IPC address format.

- Sockets over NetBIOS

  Describes how programmers can communicate with NetBIOS using the sockets API, and the NetBIOS address format.

- Windows Sockets Version 1.1 for OS/2

Presents information for implementing the Winsock 1.1 API for OS/2 applications.

- Remote Procedure Calls

  Describes the remote procedure calls and how to use them in a user-written application.

- File Transfer Protocol

  Describes the file transfer protocol routines and how to use them in a user-written application.

- Resource ReSerVation Protocol

  Describes the resource reservation protocol routines and how to use them to ensure that some quality of service can be reserved for sending and receiving on the network.

Sockets General Programming Information

This section contains technical information for planning, designing, and writing application programs that use the sockets application programming interface (API) in a TCP/IP Version 4.21 for OS/2 Warp environment.

Topics

Introduction to Networking Services
Sockets Overview
Socket Protocol Families
Socket Addresses
Socket Types
Socket Creation
Binding Names to Sockets
Socket Connections
Obtaining Socket Addresses
Server Connections
Connectionless Datagram Services
Socket Options
Socket Data Transfer
Socket Shutdown
Typical Socket Session Diagram
TCP Extensions for Transactions (T/TCP)
TCP Extensions for High Performance (RFC 1323)
High Performance Send
Passing Sockets Between Processes
Multithreading Considerations
Accessing a TCP/IP API DLL from an Intermediate DLL
Differences between OS/2 and Standard BSD Sockets
Compiling and Linking a Sockets API Application
Sample Programs

Introduction to Networking Services

OS/2 Warp has integrated networking services that provide a 32-bit sockets API for the:

- Internet (TCP/IP) domain
- NetBIOS communication domain
- Local interprocess communication (Local IPC) domain

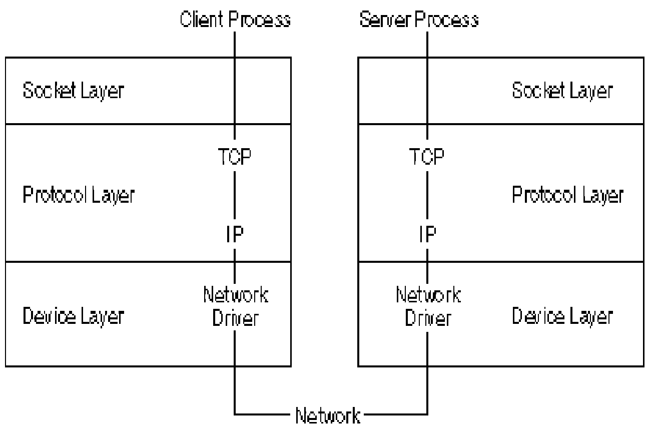
The sockets API lets you write distributed or client/server applications using TCP/IP or NetBIOS to communicate across networks. The API also allows interprocess communication within a single workstation. OS/2 Warp's sockets API is based on the Berkeley Software Distribution (BSD) Version 4.4 sockets implementation.

The OS/2 Warp operating system's networking services consists of three layers:

- The sockets layer
- The protocol layer
- The device layer

The *sockets layer* supplies the interface between the calls and lower layers, the *protocol layer* contains the protocol stacks used for communication, and the *device layer* contains the device drivers that control the network devices. The following figure illustrates the relationship between the layers:

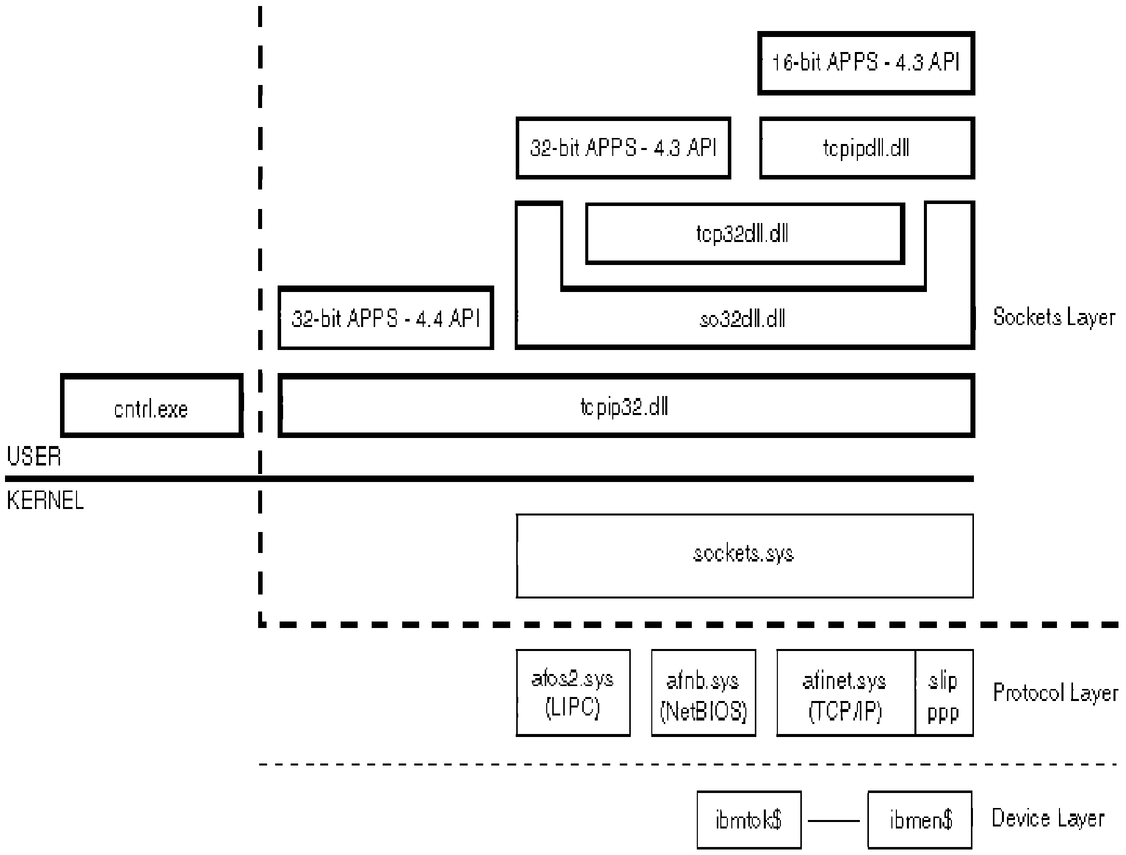
Client/Server Model



Processes communicate using the *client/server model.* In this model, a server process acting as one endpoint of a two-way communication path listens to a socket. At the other end a client process communicates to the server process through another socket. The client process can be on the same machine or on a different machine from the server process. The protocol stack(s) on the machine(s) maintains internal connections, and routes data between the client and server.

The following figure describes the OS/2 Warp kernel and internal structure of TCP/IP Version 4.21 for OS/2 Warp.

Internal Structure of TCP/IP



The major components of the OS/2 TCP/IP stack are:

Control Program

CNTRL.EXE provides threads to run the TCP/IP stack. It provides a thread for each of the following:

- TCP fast timeout processing
- TCP slow timeout processing
- Debug thread for IP
- ARP timeout processing
- Watchdog thread for the adapter status
- Loopback IP packets processing

CNTRL.EXE is normally started from CONFIG.SYS with a RUN = statement. It should be the first program to begin executing when TCP/IP is started.

Sockets Layer The sockets layer comprises the dynamic link libraries for the different categories of applications, and the device drivers.

- DLLs:

  TCPIP32.DLL exports the 32-bit BSD Version 4.4 sockets API to applications. *TCP32DLL.DLL* and *SO32DLL.DLL* together export the 32-bit BSD Version 4.3 socket APIs to applications. These three DLLs are thread-reentrant. *TCPIPDLL.DLL* provides the sockets APIs for 16-bit applications.

- Device driver:

  SOCKETS.SYS provides the common sockets layer for the protocol stacks. Calls made to the socket APIs first pass through SOCKETS.SYS, which routes the call to the correct protocol stack. The socket address families supported are *AF_OS2* (or equivalently, *AF_UNIX*), *AF_INET,* and *AF_NETBIOS* (or equivalently, *AF_NB*).

Protocol Layer The protocol layer holds the device drivers.

AFOS2.SYS is the *Local Interprocess Communication (LIPC)* device driver. This driver supports AF_OS2 and AF_UNIX socket types. These socket types can be used by applications within one OS/2 machine to communicate with each other.

AFNB.SYS device driver provides support for sockets over NetBIOS. This driver supports applications written using the AF_NETBIOS or AF_NB socket type.

AFINET.SYS is the transport protocol device driver for the AF_INET socket type. AFINET.SYS is essentially TCP/IP code. It is compliant with NDIS version 2.0.1; any MAC driver written to that specification should work with the stack. SLIP, PPP, X25, and SNAlink use a special interface in this driver to their respective hardware.

R0LIB32 Library

Ring-3 application programs access the networking kernel through tcp32dll.dll and so32dll.dll. For the device drivers at Ring-0, the ring transitions to access the networking kernel (sockets.sys) are very costly, and hence the r0lib.lib interface provides a way to access the networking kernel through 16-bit API calls. Because the r0lib is a 16-bit interface and the TCP/IP stack version 4.21 and above is a 32-bit stack, version 4.21 provides a new library at Ring-0, called **r0lib32**.

This new library not only saves the thunking in the stack kernel, but also bypasses the thunking layer, which is required for the 32-bit device drivers to go through the existing r0lib. All 32-bit device drivers can now use the **r0lib32** to get into the stack without any thunking or ring transition overheads.

Sockets Overview

This section provides some background information about the OS/2 Warp sockets API, defines it in more detail, and describes its basic functions.

Topics

Sockets Background
What Is a Socket?
Socket Facilities

Sockets Background

The sockets API was developed in response to the need for sophisticated interprocess communication facilities to meet the following goals:

- Provide access to communications networks such as an internet
- Enable communication between unrelated processes, which can either reside locally on the same host computer or on multiple host machines

The sockets API provides a generic interface that allows networking applications to use any protocol stack. After you pick the protocol stack, you can choose the type of socket that you want, based on the communication characteristics that you desire. For example, stream sockets offer a reliable method of data transmission without message boundaries, whereas datagram sockets offer message boundaries but do not guarantee reliability.

What Is a Socket?

A socket is a communication channel abstraction that enables unrelated processes to exchange data locally and across networks. A single socket is one endpoint of a full-duplex (two-way) communication channel. This means that data can be transmitted and received simultaneously. From an application program perspective, a socket is a resource allocated by the operating system, similar to a file handle. It is represented by an unsigned integer called a *socket descriptor.* A pair of sockets is used to communicate between processes on a single workstation or different workstations. Each socket of the pair is used by its own process to send and receive data with the other socket.

When a socket is created, it is associated with a particular protocol stack (called the protocol family) and socket type within that family. Communication can occur only between sockets that use the same socket type within the same protocol family.

Socket Facilities

Socket calls and network library calls provide the building blocks for IPC. An application program must perform the basic functions, described in the following sections, to conduct IPC through the sockets layer.

Topics

Creating and Binding Sockets
Accepting and Initiating Socket Connections
Sending and Receiving Data
Shutting Down Socket Operations
Translating Network Addresses

Creating and Binding Sockets

A socket is created with the socket() call. This call creates a socket of a specified:

- protocol family
- socket type
- protocol

Sockets have different qualities depending on these specifications. The *protocol family* specifies the protocol stack to be used with the created socket. The *socket type* defines its communication properties such as reliability, ordering, and prevention of duplication of messages (see Socket Types). The *protocol* specifies which network protocol to use within the domain. The protocol must support the characteristics requested by the socket type.

An application can use the bind() call to associate a local name (usually a network address) with a socket. The form and meaning of socket addresses are dependent on the protocol family in which the socket is created. The socket name is specified by a sockaddr structure.

The bind() call is optional under some circumstances: the connect() call and any of the data transmission calls (for example, send()) will automatically associate the local name to the socket if bind() hasn't been called.

Accepting and Initiating Socket Connections

Sockets can be connected or unconnected. Unconnected sockets are produced by the socket() call. An unconnected socket can become a connected socket by:

- Client application: Actively connecting to another socket, using the connect() call

- Server application: Binding a name to the socket, and then listening for and accepting a connection from another socket, using the listen() and accept() calls.

Stream sockets require a connection before you can transfer data. Other types of sockets, such as datagram sockets, need not establish connections before use.

Sending and Receiving Data

The sockets API includes a variety of calls for transferring data. They all operate similarly, but take different parameters to support different levels of functionality. Some socket calls support scatter-gather communication. Some support only connected sockets while others will work on any socket. Some calls support additional flags to control how data is sent or received. See Data Transfer Calls for a summary of which calls support which options.

Shutting Down Socket Operations

Once sockets are no longer of use they can be shut down or closed using the shutdown() or soclose() call. Typically, the shutdown() call is used if you want to shut down data transfer in one direction while keeping the other direction open. The soclose() call shuts down data transfer in both directions and then releases the resources associated with the socket.

Translating Network Addresses

Application programs need to translate human-readable addresses into the low-level form used by the protocol. The sockets API includes calls to:

- Map host names to IP addresses and back
- Map network names to numbers and back
- Map service and protocol names to numbers and back
- Convert numbers from *network-byte order* (big-endian) to *host-byte order* (which is little-endian on OS/2 machines) and back

----

Socket Protocol Families

Sockets that share common communication properties, such as naming conventions and address formats, are grouped into *protocol families.*

A protocol family includes the following:

- Rules for manipulating and interpreting network addresses
- A collection of related address formats that comprise an address family
- A set of network protocols

Topics

Supported Protocol Families
TCP/IP Properties
Local IPC Properties
NetBIOS Domain Properties
Routing Domain Properties

Supported Protocol Families

The supported domains' protocol families are defined in the <SYS\SOCKET.H> header file and are listed in the following table:

Protocol Families Supported

Protocol Family	#define in <SYS\SOCKET.H>	Supported Protocols	Supported Socket Types
TCP/IP	PF_INET	ICMP, IP, TCP, UDP	Datagram, raw, stream
Local IPC	PF_OS2 or PF_UNIX	Local IPC	Datagram, stream

NetBIOS	PF_NETBIOS or PF_NB	NetBIOS	Datagram, sequenced packet
Routing	PF_ROUTE	Routing messages	Raw

As the table indicates, some socket types can be used in more than one protocol family.

TCP/IP Properties

Provides socket communication between a local process and a process running on a remote host. The SOCK_STREAM socket type is supported by TCP (Transmission Control Protocol); the SOCK_DGRAM socket type is supported by UDP (User Datagram Protocol). Each is layered on top of the transport-level Internet Protocol (IP). ICMP (Internet Control Message Protocol) is implemented on top of IP and is accessible through a raw socket. The raw socket interface, SOCK_RAW sockets, allows access to the raw facilities of IP. Each raw socket is associated with one IP protocol number and receives all traffic for that protocol. This allows administrative and debugging functions to occur and enables user-level implementations of special-purpose protocols such as inter-gateway routing protocols.

Local IPC Properties

Provides socket communication between processes running on the same machine. The SOCK_STREAM socket type provides pipe-like facilities, while the SOCK_DGRAM socket type provides reliable message-style communications.

NetBIOS Domain Properties

Provides connection-oriented, reliable, full-duplex service to an application. SOCK_SEQPACKET provides reliable message-style communications, while SOCK_DGRAM provides a connectionless mode of communications.

Routing Domain Properties

The PF_ROUTE domain supports communications between a process and the routing facilities in the kernel.

Socket Addresses

Sockets can be named with an address so that processes can connect to them. The sockets layer treats an address as an opaque object. Applications supply and receive addresses as tagged, variable-length byte strings. A sockaddr data structure can be used as a template for referring to the identifying tag of each socket address.

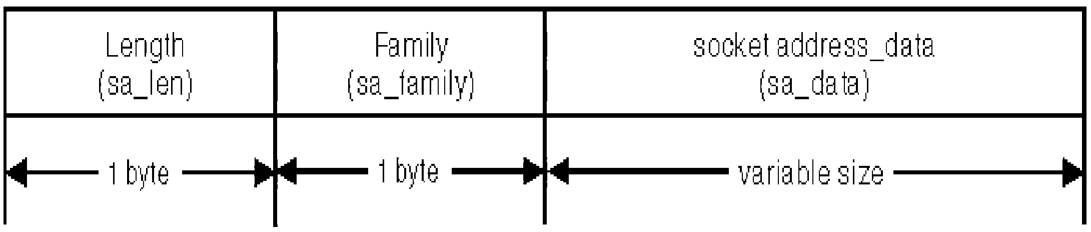
Topics

Socket Address Data Structures
Socket Addresses in TCP/IP
Connection Modes

Socket Address Data Structures

The sockaddr data structure is used to provide a generic name for a socket. The following figure illustrates this data structure:

sockaddr Structure



The <SYS\SOCKET.H> file contains this data structure. The sa_len field specifies how long the address is; this field is only used with routing sockets. The family (sa_family) identifies which protocol family this address corresponds to. The contents of the socket address data (sa_data) field depend on the protocol in use.

Additional data structures are defined that correspond to a particular protocol family and overlay the sockaddr structure. The types of socket address data structures are as follows:

Socket Address Data Structures

Data Structure	sa_family Value	Protocol	Header File
struct sockaddr_in	AF_INET	TCP/IP	<NETINET\IN.H>
struct sockaddr_un	AF_OS2 or AF_UNIX	Local IPC	<SYS\UN.H>
struct sockaddr_nb	AF_NETBIOS or AF_NB	NetBIOS	<NETNB\NB.H>

The contents of the various socket addresses are as follows:

Internet
: A socket name in the internet domain is an internet address, made up of a 32-bit IP address and a 16-bit port number. The 32-bit address is composed of network and host parts; the network part is variable in size. The host part can be interpreted optionally as a subnet field plus the host on a subnet; this is enabled by setting a network address mask.

OS/2 or UNIX
: A socket name in the OS/2 or UNIX domain is a unique listing of ASCII characters of up to 108 bytes.

NetBIOS
: A socket name in the NetBIOS domain is made up of a 16-byte NetBIOS name and is used as is.

Socket Addresses in TCP/IP

TCP/IP provides a set of 16-bit port numbers within each host. Because each host assigns port numbers independently, it is possible for sockets on different hosts to have the same port number.

To ensure that socket addresses are unique within a network, TCP/IP concatenates the internet address of the LAN interface with the port number to devise the internet socket address. Since a host's internet address is always unique within a network, the socket address for a particular socket on a particular host is unique. Additionally, since each connection is fully specified by the pair of sockets it joins, every connection between internet hosts is also uniquely identified.

The port numbers from 0 to 1023 are reserved for official internet services. Port numbers in the range of 1024-49151 are reserved for other registered services that are common on internet networks. These port numbers are listed in the ETC\SERVICES file. When a client process needs one of these well-known services at a particular host, the client process sends a service request to the socket address for the well-known port at the server.

The port numbers from 49152 to 65535 are generally used by client processes which need a port, but don't care which one they get. These port numbers are usually assigned by the TCP/IP stack when a connect() or sendto() is performed without having done a previous bind().

Connection Modes

A connection mode refers to an established logical channel for the transmission of data between two application programs.

The sockets API supports two connection modes:

- Connection-oriented
- Connectionless

In the connectionless mode, sockets are not tied to a destination address. Applications sending messages can specify a different destination address for each datagram, if necessary, or they can tie the socket to a specific destination address for the duration of the connection.

The connection-oriented mode requires a logical connection to be established between two applications before data transfer or communication can occur. Applications encounter some overhead during the connection establishment phase as the applications negotiate the connection request. This mode is useful for applications that use long datastream transmissions or require reliable transmissions of data.

The connectionless mode does not require a logical connection to allow communication between applications. Rather, individual messages are transmitted independently from one application to another application. Each message must contain the data and all information required for delivery of the message to its destination. Normally, datagram and raw socket types use the connectionless mode.

The term *connected* refers to two endpoints that have an established logical connection between them. Stream and sequenced packet socket types use the connection-oriented mode. For information on how datagram and raw socket types can be *connected*, see Datagram or Raw Sockets.

Socket Types

Sockets are classified according to communication properties. Processes usually communicate between sockets of the same type. However, if the underlying communication protocols support the communication, sockets of different types can communicate.

Each socket has an associated type, which describes the semantics of communications using that socket. The socket type determines the socket communication properties such as reliability, ordering, and prevention of duplication of messages. The basic set of socket types is defined in the <SYS\SOCKET.H> file:

```
/*Standard socket types */

#define  SOCK_STREAM            1 /*virtual circuit*/

#define  SOCK_DGRAM             2 /*datagram*/

#define  SOCK_RAW               3 /*raw socket*/

#define  SOCK_SEQPACKET         5 /*sequenced packet stream*/
```

Other socket types can be defined.

OS/2 supports the basic set of sockets:

SOCK_DGRAM

Provides datagrams, which are connectionless messages of a fixed maximum length. This type of socket is generally used for short messages, such as a name server or time server, since the order and reliability of message delivery is not guaranteed.

A *datagram* socket supports the bidirectional flow of data which is not sequenced, reliable, or unduplicated. A process receiving messages on a datagram socket may find messages duplicated or in an order different from the order sent. Record boundaries in data are, however, preserved. Datagram sockets closely model the facilities found in many contemporary packet-switched networks.

An application can use the sendto() and recvfrom() calls or the sendmsg() and recvmsg() calls to exchange data over a datagram socket. If an application is using datagram sockets and calls connect() fully specifying the destination address, the socket will be considered *connected*. The application program can then use the other data transfer calls send() and recv() or writev() and readv(). The connected or unconnected method of data transfer stays in effect until connect() is called again with a different destination address.

Datagram sockets may be used to send broadcast messages over TCP/IP and NetBIOS. For TCP/IP, the constant INADDR_BROADCAST, defined in <NETINET\IN.H>, can be used to send a broadcast datagram. For NetBIOS, the address format has a type field that specifies whether the address is unique, multicast, or broadcast.

SOCK_STREAM

Provides sequenced, two-way byte streams with a transmission mechanism for stream data. This socket type transmits data reliably (in order, not duplicated, and retransmitted if necessary) without record boundaries, and with out-of-band capabilities.

There is no guarantee for a one-to-one correspondence of send and receive calls. It is possible for data sent by one send() call to be received by more than one different receive call, or the other way around.

Stream sockets are either active or passive. Active sockets are used by clients who *actively* initiate connection requests with connect(). Passive sockets are used by servers to *passively* wait for and accept connection requests with the listen() and accept() calls. A passive socket that has indicated its willingness to accept connections with the listen() call cannot be used to initiate connection requests.

After a connection has been established between stream sockets, any of the data transfer calls can be used:

- send() and recv()
- sendto() and recvfrom()
- sendmsg() and recvmsg()
- writev() and readv()

Usually, a send()-recv() pair is used for sending data on stream sockets.

SOCK_RAW

Provides access to internal network protocols and interfaces. A raw socket allows an application direct access to lower-level communication protocols, such as IP and ICMP. Raw sockets are intended for advanced users who wish to take advantage of some protocol feature not directly accessible through a normal interface, or who wish to build new protocols atop existing low-level protocols.

Raw sockets are normally datagram-oriented, though their exact characteristics are dependent upon the interface provided by the protocol. However, raw sockets can be connected if connect() is called to specify the destination address.

SOCK_SEQPACKET

Provides sequenced, reliable, and unduplicated flow of information. Every sequenced packet is sent and received as a complete record.

After a connection has been established between sequenced packet sockets, any of the data transfer calls can be used:

- send() and recv()
- sendto() and recvfrom()
- sendmsg() and recvmsg()
- writev() and readv()

Usually, a send()-recv() pair is used for sending data on sequenced packet sockets.

Topics

Socket Types Summary
Guidelines for Using Socket Types

Socket Types Summary

The following table summarizes many of the attributes and features of supported socket types:

Socket Types

Socket Type	#define in <SYS\SOCKET.H>	Protocols	Connection Oriented?	Primary Socket Calls
Stream	SOCK_STREAM	TCP/IP, Local IPC	yes	send() or recv()
Sequenced packet	SOCK_SEQPACKET	NetBIOS	yes	send() or recv()

Datagram	SOCK_DGRAM	TCP/IP, Local IPC, NetBIOS	no*	sendto() or recvfrom()*
Raw	SOCK_RAW	TCP/IP	no*	sendto() or recvfrom()*

Table Note (*) Datagram sockets and raw sockets are connectionless, unless the application has called connect() for the socket. In this case, the socket is **connected**. Refer to Connection Modes for additional information.

Guidelines for Using Socket Types

If you are communicating with an existing application, you must use the same socket type and the same protocol as the existing application.

Raw sockets have a special purpose of interfacing directly to the underlying protocol layer. If you are writing a new protocol on top of Internet Protocol (IP) or wish to use the Internet Control Message Protocol (ICMP), then you must use raw sockets.

You should consider the following factors in choosing a socket type for new applications:

- Reliability: Stream and sequenced packet sockets provide the most reliable connection. Connectionless datagram and raw sockets are unreliable because packets can be discarded, duplicated, or received out of order. This may be acceptable if the application does not require reliability, or if the application implements the reliability on top of the sockets API.

- Performance: The overhead associated with reliability, flow control, packet reassembly, and connection maintenance degrades the performance of stream and sequenced packet sockets so that these socket types do not perform as well as datagram sockets acting in a connectionless mode.

- Amount of data to be transferred: Datagram and sequenced packet sockets impose a limit on the amount of data transferred in each packet.

Socket Creation

The basis for communication between processes centers on the socket mechanism. A socket is comparable to an OS/2 file handle. Application programs request the operating system to create a socket through the use of the socket() call. When an application program requests the creation of a new socket, the operating system returns an integer that the application program uses to reference the newly created socket.

To create a socket with the socket() call, the application program must include a protocol family and a socket type. It can also include a specific communication protocol within the specified protocol family.

An example of an application using the socket() call is:

An Application Using the socket() Call

```
int s;
...
s = socket(PF_INET, SOCK_STREAM, 0);
```

In this example, the socket() call allocates a socket descriptor *s* in the internet protocol family (PF_INET). The *type* parameter is a constant that specifies the type of socket. For the internet communication domain, this can be SOCK_STREAM, SOCK_DGRAM, or SOCK_RAW. The *protocol* parameter is a constant that specifies which protocol to use. Passing 0 chooses the default protocol for the specified socket type. Supported Protocol Families includes information on default protocols. If successful, socket() returns a non-negative integer socket descriptor.

See Socket Connections for more about creating sockets.

Binding Names to Sockets

The socket() call creates a socket without a name. An unnamed socket is one without any association to a local address. Until a name is bound to a socket, no messages can be received on it.

Communicating processes are bound by an association. The bind() call allows a process to specify half of an association: local address and local port (TCP/IP), or local path name (NetBIOS and local IPC). The connect() and accept() calls are used to complete a socket's association.

An application program may not care about the local address it uses and may allow the protocol software to select one. This is not true for server processes. Server processes that operate at a well-known port need to be able to specify that port to the system.

In most domains, associations must be unique. Internet domain associations must never include duplicate protocol, local address, local port, foreign address, or foreign port tuples.

Wildcard addressing is provided to aid local address binding in the Internet domain. When an address is specified as INADDR_ANY (a constant defined in the <NETINET\IN.H> file), the system interprets the address as any valid address.

Sockets with wildcard local addresses may receive messages directed to the specified port number and sent to any of the possible addresses assigned to a host. If a server process wished to connect only hosts on a given network, it would bind the address of the hosts on the appropriate network.

A local port can be specified or left unspecified (denoted by 0), in which case the system selects an appropriate port number for it.

The bind() call accepts the *s, name,* and *namelen* parameters. The *s* parameter is the integer descriptor of the socket to be bound. The *name* parameter specifies the local address, and the *namelen* parameter indicates the length of address in bytes. The local address is defined by a data structure termed sockaddr.

In the internet domain, a process does not have to bind an address and port number to a socket, because the connect() and send() calls automatically bind an appropriate address if they are used with an unbound socket.

The bound name is a variable-length byte string that is interpreted by the supporting protocols. Its interpretation can vary from protocol family to protocol family (this is one of the properties of the protocol family).

An example of an application using the bind() call is:

An Application Using the bind() Call

```
int rc;
int s;
struct sockaddr_in myname;

/* clear the structure */
memset(&myname, 0, sizeof(myname));
myname.sin_len = sizeof(myname);
myname.sin_family = AF_INET;
myname.sin_addr.s_addr = inet_addr("129.5.24.1"); /* specific interface */
myname.sin_port = htons(1024);
...
rc = bind(s, (struct sockaddr *) &myname, sizeof(myname));
```

For a server in the internet domain to be able to listen for connections on a stream socket or issue recvfrom() on a datagram socket, the server must first bind the socket to a specific address family, local address, and local port. This example binds *myname* to socket *s*. Note that the sockaddr_in structure should be zeroed before calling bind(). For a more detailed description, see bind(). For information on the sockaddr_in structure, see Internet Address Formats.

The unique name *myname* specifies that the application uses an internet address family (AF_INET) at internet address 129.5.24.1, and is bound to port 1024. The preceding example shows two useful network utility routines.

- inet_addr() takes an internet address in dotted-decimal form and returns it in network-byte order. For a more detailed description, see inet_addr().
- htons() takes a port number in host-byte order and returns the port in network-byte order. For a more detailed description, see htons().

The next figure shows how the bind() call on the server side uses the network utility routine getservbyname() to find a *well-known* port number for a specified service from the ETC\SERVICES file (for more information on well-known ports, see Ports). The figure also shows the use of the internet address wildcard value INADDR_ANY. This is the value generally used on a socket bind() call. It binds the socket to all internet addresses available on the local machine, without requiring the program to know the local internet address. (The code fragment in the preceding figure will run successfully only on the machine with internet address 192.5.24.1.) If a host has more than one internet address (that is, if it is multihomed host), messages sent to any of the addresses will be deliverable to a socket bound to INADDR_ANY.

A bind() Call Using the getservbyname() Call

```
int rc;
int s;
struct sockaddr_in myname;
struct servent *sp;
...
sp = getservbyname("login","tcp");  /* get application specific */
/* well-known port           */
...
/* clear the structure */
memset(&myname, 0, sizeof(myname));
myname.sin_len = sizeof(myname);
myname.sin_family = AF_INET;
myname.sin_addr.s_addr = INADDR_ANY;  /* any interface */
myname.sin_port = sp->s_port;
...
rc = bind(s,(struct sockaddr *)&myname,sizeof(myname));
```

See bind() for more on this call.

Socket Connections

Initially, a socket is created in the unconnected state, meaning the socket is not associated with any foreign destination. The connect() call binds a permanent destination to a socket, placing it in the connected state. An application program must call connect() to establish a connection before it can transfer data through a reliable stream socket. Sockets used with connectionless datagram services need not be connected before they are used, but connecting sockets makes it possible to transfer data without specifying the destination each time.

The semantics of the connect() call depend on the underlying protocols. In the case of TCP, the connect() call builds a TCP connection with the destination, or returns an error if it cannot. In the case of connectionless services, the connect() call does nothing more than store the destination address locally.

Connections are established between a client process and a server process. In a connection-oriented network environment, a client process initiates a connection and a server process receives, or responds to, a connection. The client and server interactions occur as follows:

- The server, when willing to offer its advertised services, binds a socket to a well-known address associated with the service.
- The server process socket is then marked to indicate incoming connections are to be accepted on it. It is then possible for an unrelated process to rendezvous with the server.
- The client requests services from the server by initiating a connection to the server's socket. The client process uses a connect() call to initiate a socket connection.
- If the client process' socket is unbound at the time of the connect() call, the system automatically selects and binds a name to the socket if necessary. This is the usual way that local addresses are bound to a socket.
- If the connection to the server fails, the client's connect() call fails (any name automatically bound by the system, however, remains). Otherwise, the socket is associated with the server and data transfer can begin.

An example of a client application using the connect() call to request a connection is:

An Application Using the connect() Call

```
int s;
struct sockaddr_in servername;
int rc;
...
memset(&servername, 0, sizeof(servname));
servname.sin_len = sizeof(servname);
servername.sin_family = AF_INET;
servername.sin_addr.s_addr = inet_addr("129.5.24.1");
servername.sin_port = htons(1024);
...
rc = connect(s, (struct sockaddr *) &servername, sizeof(servername));
```

The connect() call attempts to connect socket *s* to the server with name supplied in *servername* . This could be the server that was used in the previous bind() example. With TCP stream sockets the caller blocks until the connection is accepted by the server. On successful return from connect(), the socket *s* is associated with the connection to the server. See ioctl() for additional information about determining blocking and nonblocking behavior. Note that the sockaddr_in structure should be cleared before calling connect(). For a more detailed description, see connect().

The following figure shows an example of using the gethostbyname() network utility routine to find out the internet address of *serverhost* from the name server or the ETC\HOSTS file:

An Application Using the gethostbyname() Call

```
int s;
struct sockaddr_in servername;
char *hostname = "serverhost";
int rc;
int connect(int s, struct sockaddr *name, int namelen);  /* extracted from sys\socket.h */
struct hostent *hp;
...

hp = gethostbyname(hostname);

/* clear the structure */
memset(&servername, 0, sizeof(servname));
servername.sin_len = sizeof(servername);
servername.sin_family = AF_INET;
servername.sin_addr.s_addr = *((u_long *)hp->h_addr);
servername.sin_port = htons(1024);
...
rc = connect(s,(struct sockaddr *)&servername,sizeof(servername));
```

Obtaining Socket Addresses

The sockets API includes calls that allow an application to obtain the address of the destination to which a socket connects and the local address of a socket. The socket calls that allow a program to retrieve socket addresses are:

- getsockname()
- getpeername()

For additional information that you may need before binding or obtaining socket addresses, see:

- Socket Addresses
- Socket Connections

Server Connections

In the internet domain, the server process creates a socket, binds it to a well-known protocol port, and waits for requests. The listen() call allows server processes to prepare a socket for incoming connections. In terms of underlying protocols, the listen() call puts the socket in a passive mode ready to accept connections. When the server process starts the listen() call, it also informs the operating system that the protocol software should queue multiple simultaneous connection requests that arrive at a socket. The listen() call includes a parameter that allows a process to specify the length of the connection queue for that socket. If the queue is full when a connection request arrives, the operating system refuses the connection. The listen() call applies only to sockets that have selected reliable stream delivery or connection-oriented datagram service. An example of a server application using the listen() call is:

An Application Using the listen() Call

```
int s;
int backlog;
int rc;
...
rc = listen(s, 5);
```

The listen() call is used to indicate that the server is ready to begin accepting connections. In this example, a maximum of five connection requests can be queued for the server. Additional requests are ignored. For a more detailed description, see listen().

Once a socket has been set up, the server process needs to wait for a connection. The server process waits for a connection by using the accept() call. A call to the accept() call blocks until a connection request arrives. When a request arrives, the operating system returns the address of the client process that has placed the request. The operating system also creates a new socket that has its destination connected to the requesting client process and returns the new socket descriptor to the calling server process. The original socket still has a wildcard foreign destination which remains open.

When a connection arrives, the call to accept() returns. The server process can either handle requests interactively or concurrently. In the interactive approach, the thread that did the accept() handles the request itself, closes the new socket, and then starts the accept() call to obtain the next connection request. In the concurrent approach, after the call to accept() returns, the server process creates a new thread to handle the request. The new thread proceeds to service the request, and then exits. The original thread invokes the accept() call to obtain the next connection request.

An example of a server application for accepting a connection request by using the accept() call is:

An Application Using the accept() Call

```
int clientsocket;
int s;
struct sockaddr clientaddress;
int addrlen;
...
addrlen = sizeof(clientaddress);
...
clientsocket = accept(s, &clientaddress, &addrlen);
```

Handling Multiple Sockets

Applications can handle multiple sockets. In such situations, you can use the select() call to determine the sockets that have data to be read, those that are ready for data to be written, and the sockets that have pending exception conditions. If the timeout parameter is positive, select() waits up to the amount of time indicated for at least one socket to become ready on the indicated conditions. This is useful for applications servicing multiple connections that cannot afford to block waiting for data on one connection.

There are two versions of the select() call: a TCP/IP Version 4.21 for OS/2 Warp version and a version modeled after the BSD select() call. An example of how the TCP/IP Version 4.21 for OS/2 Warp select() call is used is shown in the next figure. For information on using the BSD version of the select() call, see select().

An Application Using the select() Call

```
...
int socks[3];    /* array of sockets */
long timeout = MAX_TIMEOUT;
int s1, s2, s3;
int number_ready;
...
/* put sockets to check in socks[] */
socks[0] = s1;     /* read socket number */
socks[1] = s2;     /* write socket number */
socks[2] = s3;     /* second write socket number */

/* check for READ on s1, WRITE on s2 and s3 */
number_ready = os2_select(socks, 1, 2, 0, timeout);
```

In this example, the application indicates the sockets to be checked for readability or readiness for writing.

Connectionless Datagram Services

The operating system provides support for connectionless interactions typical of the datagram facilities found in packet-switched networks. A datagram socket provides a symmetric interface for data exchange. While processes are still likely to be client and server, there is no requirement for connection establishment. Instead, each message includes the destination address.

An application program can create datagram sockets using the socket() call. In the internet domain, if a particular local address is needed, a bind() call must precede the first data transmission. Otherwise, the operating system sets the local address or port when data is first sent. The application program uses the sendto() and recvfrom() calls to transmit data; these calls include parameters that allow the client process to specify the address of the intended recipient of the data.

In addition to the sendto() and recvfrom() calls, datagram sockets can also use the connect() call to associate a socket with a specific destination address. In this case, any data sent on the socket is automatically addressed to the connected peer socket, and only data received from that peer is delivered to the client process. Only one connected address is permitted for each socket at one time; a second connect() call changes the destination address.

A connect() call request on a datagram socket results in the operating system recording the peer socket's address (as compared to a stream socket, where a connect request initiates establishment of an end-to-end connection). The accept() and listen() calls are not used with datagram sockets.

While a datagram socket is connected, errors from recent send() calls can be returned asynchronously. These errors can be reported on subsequent operations on the socket, or a special socket option, SO_ERROR. This option, when used with the getsockopt() call, can be used to interrogate the error status. A select() call for reading or writing returns true when a process receives an error indication. The next operation returns the error, and the error status is cleared.

See Socket Types for more information that you may need before connecting sockets.

Socket Options

In addition to binding a socket to a local address or connecting it to a destination address, application programs need a method to control the behavior of a socket. For example, when using protocols that use timeout and retransmission, the application program may want to obtain or set the timeout parameters. An application program may also want to control the allocation of buffer space, determine if the socket allows transmission of broadcast, or control processing of out-of-band data. The ioctl-style getsockopt() and setsockopt() calls provide the means to control socket operations. The getsockopt() call allows an application program to request information about socket options. The setsockopt() call allows an application program to set a socket option using the same set of values obtained with the getsockopt() call. Not all socket options apply to all sockets. The options that can be set depend on the current state of the socket and the underlying protocol being used.

Socket Data Transfer

Most of the work performed by the sockets layer is in sending and receiving data. The sockets layer itself does not impose any structure on data transmitted or received through sockets. Any data interpretation or structuring is logically isolated in the implementation of the protocol family.

Once a connection is established between sockets, an application program can send and receive data.

Topics

Sending and Receiving Data
Out-of-Band Data
Socket I/O Modes

Sending and Receiving Data

Sending and receiving data can be done with any one of several calls. The calls vary according to the amount of information to be transmitted and received and the state of the socket being used to perform the operation.

- The writev() call can be used with a socket that is in a connected state, as the destination of the data is implicitly specified by the connection.
- The sendto() and sendmsg() calls allow the process to specify the destination for a message explicitly.
- The recv() call allows a process to receive data on a connected socket without receiving the sender's address.
- The recvfrom() and recvmsg() calls allow the process to retrieve the incoming message and the sender's address.

While the send() and recv() calls are virtually identical to the readv() and writev() calls, the extra flags argument in the send() and recv() calls is important. The flags, defined in the <SYS\SOCKET.H> file, can be defined as a nonzero value if the application program requires one or more of the following:

MSG_OOB	Sends or receives out-of-band data.
MSG_PEEK	Looks at data without reading.
MSG_DONTROUTE	Sends data without routing packets.

Out-of-band data is specific to stream sockets. The option to have data sent without routing applied to the outgoing packets is currently used only by the routing table management process, and is unlikely to be of interest to the casual user. The ability to preview data is, however, of general interest. When the MSG_PEEK flag is specified with a recv() call, any data present is returned to the user, but treated as still unread. That is, the next readv() or recv() call applied to the socket returns the data previously previewed.

The following example shows an application sending data on a connected socket and receiving data in response. The *flags* field can be used to specify additional options to send() or recv(), such as sending out-of-band data. For additional information, see send() and recv().

An Application Using the send() and recv() Calls

```
int bytes_sent;
int bytes_received;
char data_sent[256] = "data to be sent on connected socket";
char data_received[256];
int s;
...
bytes_sent = send(s, data_sent, sizeof(data_sent), 0);
...
bytes_received = recv(s, data_received, sizeof(data_received), 0);
```

An example of an application sending from a connected socket and receiving data in response is:

An Application Using the sendto() and recvfrom() Call

```
int bytes_sent;
int bytes_received;
char data_sent[256] = "data to be sent using sendto()";
char data_received[256];
struct sockaddr_in to;
struct sockaddr from;
int addrlen;
int s;
...
memset(&to, 0, sizeof(to));
to.sin_len = sizeof(to);
to.sin_family = AF_INET;
to.sin_addr.s_addr   = inet_addr("129.5.24.1");
to.sin_port   = htons(1024);
...
bytes_sent = sendto(s, data_sent, sizeof(data_sent), 0, (struct sockaddr *) &to, sizeof(to));
...
addrlen = sizeof(from); /* must be initialized */
bytes_received = recvfrom(s, data_received, sizeof(data_received), 0, &from, &addrlen);
```

The sendto() and recvfrom() calls take additional parameters that allow the caller to specify the recipient of the data or to be notified of the sender of the data. See recvfrom(), and sendto(), for more information about these additional parameters.

A list of the data transfer calls and a summary of some of their characteristics follows:

Data Transfer Calls

Data Transfer Call	Buffers	Option Flags?	Sockets Used With	Server Address Required?
send()	Single	Yes	Connected only	No
recv()	Single	Yes	Connected only	No
sendto()	Single	Yes	Any socket	Yes
recvfrom()	Single	Yes	Any socket	Yes
writev()	Multiple	No	Connected only	No
readv()	Multiple	No	Connected only	No
sendmsg()	Multiple	Yes	Any socket	Yes
recvmsg()	Multiple	Yes	Any socket	Yes

For additional information that you may need when obtaining or setting socket options, see:

- Socket Types
- Out-of-Band Data
- IP Multicasting

Out-of-Band Data

The stream socket abstraction includes the concept of out-of-band data. *Out-of-band* data is a logically independent transmission channel associated with each pair of connected stream sockets. Out-of-band data can be delivered to the socket independently of the normal receive queue or within the receive queue depending upon the status of the SO_OOBINLINE socket-level option. The abstraction defines that the out-of-band data facilities must support the reliable delivery of at least one out-of-band message at a time. This message must contain at least one byte of data, and at least one message can be pending delivery to the user at any one time.

For communication protocols that support only in-band signaling (that is, the urgent data is delivered in sequence with the normal data), the operating system normally extracts the data from the normal data stream and stores it separately. This allows users to choose between receiving the urgent data in order and receiving it out of sequence without having to buffer all the intervening data.

If multiple sockets have out-of-band data awaiting delivery, an application program can use a select() call for exceptional conditions to determine those sockets with such data pending. The select() call does not indicate the actual arrival of the out-of-band data, but only notification that it is pending.

In addition to the information passed, a logical mark is placed in the data stream to indicate the point at which the out-of-band data was sent. When a signal flushes any pending output, all data up to the mark in the data stream is discarded.

To send an out-of-band message, the MSG_OOB flag is supplied to a send() or sendto() call. To receive out-of-band data, an application program must set the MSG_OOB flag when performing a recvfrom() or recv() call.

An application program can determine if the read pointer is currently pointing at the logical mark in the data stream, by using the SIOCATMARK ioctl() call.

A process can also read or peek at the out-of-band data without first reading up to the logical mark. This is more difficult when the underlying protocol delivers the urgent data in-band with the normal data, and only sends notification of its presence ahead of time (that is, the TCP protocol used to implement streams in the internet domain). With such protocols, the out-of-band byte may not have arrived when a recv() call is performed with the MSG_OOB flag. In that case, the call will return an SOCEWOULDBLOCK error code. There may be enough in-band data in the input buffer for normal flow control to prevent the peer from sending the urgent data until the buffer is cleared. The process must then read enough of the queued data so that the urgent data can be delivered.

Certain programs that use multiple bytes of urgent data, and that must handle multiple urgent signals, need to retain the position of urgent data within the stream. The socket-level option, SO_OOBINLINE provides the capability. With this option, the position of the urgent data (the logical mark) is retained. The urgent data immediately follows the mark within the normal data stream that is returned without the MSG_OOB flag. Reception of multiple urgent indications causes the mark to move, but no out-of-band data is lost.

Socket I/O Modes

Sockets can be set to either blocking or nonblocking I/O mode. The FIONBIO ioctl operation is used to determine this mode. When the FIONBIO ioctl is set, the socket is marked *nonblocking*. If a read is tried and the desired data is not available, the socket does not wait for the data to become available, but returns immediately with the SOCEWOULDBLOCK error code.

When the FIONBIO ioctl is not set, the socket is in *blocking mode*. In this mode, if a read is tried and the desired data is not available, the calling process waits for the data. Similarly, when writing, if FIONBIO is set and the output queue is full, an attempt to write causes the process to return immediately with an error code of SOCEWOULDBLOCK.

An example of using the ioctl() call to help perform asynchronous (nonblocking) socket operations is:

An Application Using the ioctl() Call

```
int s;
int bytes_received;
int dontblock;
char buf[256];
int rc;
...
dontblock = 1;
...
rc = ioctl(s, FIONBIO, (char *) &dontblock);
...
bytes_received = recv(s, buf, sizeof(buf), 0);
if (bytes_received == -1)
{
if (sock_errno() == SOCEWOULDBLOCK)
/* data is not present */
else
/* error occurred */
}
else
/* bytes_ received indicates amount of data received in buf */
```

This example causes the socket *s* to be placed in nonblocking mode. When this socket is passed as a parameter to calls that would block, such as recv() when data is not present, it causes the call to return with an error code, and sets the error value to SOCEWOULDBLOCK. Setting the mode of the socket to be nonblocking allows an application to continue processing without becoming blocked. For a more detailed description, see ioctl().

When performing nonblocking I/O on sockets, a program must check for the SOCEWOULDBLOCK error code. This occurs when an operation would normally block, but the socket it was performed on is marked as nonblocking. The following socket calls return a SOCEWOULDBLOCK error code:

- accept()
- send()
- recv()
- readv()
- writev()

Processes using these calls should be prepared to deal with the SOCEWOULDBLOCK error code. For a nonblocking socket, the connect() call returns an SOCEINPROGRESS error code if the connection cannot be completed immediately.

If an operation such as a send operation cannot be done completely, but partial writes are permissible (for example when using a stream socket), the data that can be sent immediately is processed, and the return value indicates the amount actually sent.

Socket Shutdown

Once a socket is no longer required, the calling program can discard the socket by applying a soclose() call to the socket descriptor. If a reliable delivery socket has data associated with it when a close takes place, the system continues to attempt data transfer. However, if the data is still not delivered, the system discards the data. Should the application program have no use for any pending data, it can use the shutdown() call on the socket prior to closing it.

Topics

Closing Sockets

Closing Sockets

Closing a socket and reclaiming its resources is not always a straightforward operation. In certain situations, such as when a process exits, a soclose() call is expected to be successful. However, when a socket promising reliable delivery of data is closed with data still queued for transmission or awaiting acknowledgment of reception, the socket must attempt to transmit the data. If the socket discards the queued data to allow the soclose() call to complete successfully, it violates its promise to deliver data reliably. Discarding data can cause naive processes, which depend upon the implicit semantics of the soclose() call, to work unreliably in a network environment. However, if sockets block until all data has been transmitted successfully, in some communication domains a soclose() call may never complete.

The sockets layer compromises in an effort to address this problem and maintain the semantics of the soclose() call. In normal operation, closing a socket causes any queued but unaccepted connections to be discarded. If the socket is in a connected state, a disconnect is initiated. When the disconnect request completes, the network support notifies the sockets layer, and the socket resources are reclaimed. The network layer may then attempt to transmit any data queued in the socket's send buffer, although this is not guaranteed.

Alternatively, a socket may be marked explicitly to force the application program to linger when closing until pending data are flushed and the connection has shut down. This option is marked in the socket data structure using the setsockopt() call with the SO_LINGER option. The setsockopt() call, using the linger option, takes a linger structure. When an application program indicates that a socket is to linger, it also specifies a duration for the lingering period. If the lingering period expires before the disconnect is completed, the socket layer forcibly shuts down the socket, discarding any data still pending.

An example of deallocating the socket descriptor s using the soclose() call is:

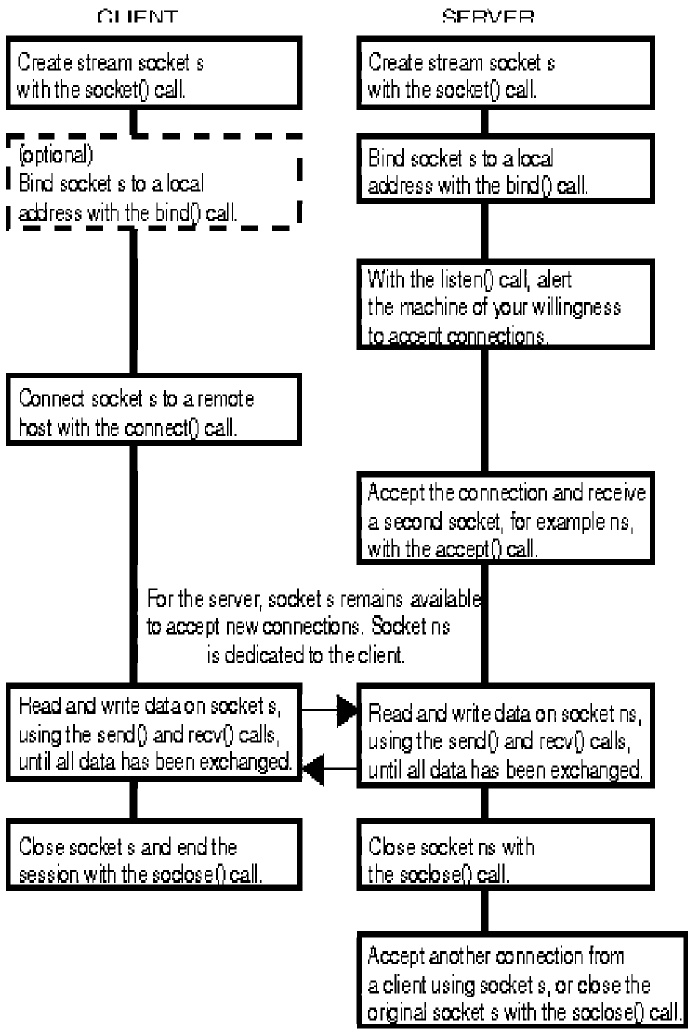
An Application Using the soclose() Call

```
...
/* close the socket */
soclose(s);
...
```

Typical Socket Session Diagram

The following figure shows a graphical representation for the general sequence of socket calls needed to provide communication between applications for supported socket types. For stream or sequenced packet socket types, and for datagram socket types, see the following figure. This basic sequence is the same for each supported protocol family for all supported socket types. This means that a programmer can modify the protocol family selection and the networking addressing parameters of an existing sockets program, recompile and relink, and the program can be run with another protocol. This also allows programs that use sockets with multiple protocols to be easily constructed.

A Typical Stream or Sequenced Packet Socket Session



CLIENT
SERVER
Create stream socket s with the socket() call.
Create stream socket s with the socket() call.
(optional) Bind socket s to a local address with the bind() call.
Bind socket s to a local address with the bind() call.
With the listen() call, alert the machine of your willingness to accept connections.
Connect socket s to a remote host with the connect() call.
Accept the connection and receive a second socket, for example ns, with the accept() call.
For the server, socket s remains available to accept new connections. Socket ns is dedicated to the client.
Read and write data on socket s, using the send() and recv() calls, until all data has been exchanged.
Read and write data on socket ns, using the send() and recv() calls, until all data has been exchanged.
Close socket s and end the session with the soclose() call.
Close socket ns with the soclose() call.
Accept another connection from a client using socket s, or close the original socket s with the soclose() call.

TCP Extensions for Transactions (T/TCP)

A client-server transaction is a client request to a server followed by the server's reply. TCP Extensions for Transactions (T/TCP) is an extension to TCP designed to make client-server transactions more efficient. For typical transaction processing, a reliable delivery of data is needed, with no explicit connection setup or tear down, and with minimal idle state at both ends. UDP is faster but not reliable. Standard TCP provides the reliability but with the overhead of connection setup and time wait delay. T/TCP addresses these needs. T/TCP can match UDP performance, and adds reliability.

The goal of T/TCP is to allow each transaction (request and response sequence) to be efficiently performed as a single incarnation of a TCP connection. Standard TCP imposes two performance problems for transaction-oriented communication. First, a TCP connection is opened with a three-way handshake which must complete successfully before any data can be transferred. This handshake adds an extra round-trip time to transaction latency. Second, closing a TCP connection leaves one or both ends in time wait state (2 maximum segment lifetime periods (2 MSL)), which limits the rate of successive transactions between the same host-port pair, since a new incarnation cannot be reopened until the time wait delay expires.   (The default value for the time wait delay can be checked by using the INETCFG.EXE utility.)

T/TCP addresses the handshake issue by using TCP Accelerated Open (TAO), which is based on using a new incarnation number called 'connection count (CC)'. T/TCP uses the monotonic property of CC values to bypass the handshake. A T/TCP server host keeps a cache containing the last valid CC value that it has received from each different client host. If an initial SYN segment (connection request) from a particular client host carries a CC value larger than the cached value, then the incoming segment is ensured to be new and the connection can be accepted immediately. The use of CC also truncates the time wait delay. Apart from rate of transactions, reduction in the time wait delay also saves locked-up resources from the hosts.

T/TCP introduces three new TCP options; namely, CC, CCnew and CCecho, and adds seven new states to the existing ten finite TCP states. To invoke T/TCP, both client and server ends of the transaction must support T/TCP. The CC value is sent by the client in the SYN segment (for active open) indicating a willingness to use T/TCP. The server sends back CCecho in the SYN-ACK segment, declaring that it understands T/TCP. The client can still optionally disable T/TCP and force a handshake by sending a CCNew segment. If any of the ends do not support T/TCP, the connection procedure reverts to the normal TCP connection setup procedure.

To take advantage of T/TCP, the socket calls sequence on a typical T/TCP client application is different from the client shown in A Typical Stream or Sequenced Packet Socket Session. On a typical T/TCP client application, after opening a Stream socket the client does not call connect(). Instead, it calls sendto() with the flag of MSG_EOF. This will send the request to the server, establish the connection, and cause a FIN to be sent from the client to the server. This single sendto() call combines the functionality of connect(), write(), and shutdown() of a normal TCP session. The typical T/TCP transaction server application is the same as a typical TCP server, except the MSG_EOF flag can be indicated on the send() call to cause a FIN to be sent at the end of the transaction data.

T/TCP can be turned on for a system-wide effect through the INET configuration utility command 'inetcfg -set CC 1'. You can use getsockopt() and setsockopt() for each socket through the TCP_CC socket option. By default the T/TCP option is turned off.

To assure that T/TCP is being invoked, IP tracing function can be turned on through 'iptrace -i', and the trace file IPTRACE.DMP can be formatted with the IPFORMAT utility, which can explicitly show the three CC option values in the formatted dump.

For more details on T/TCP functional specification, refer to RFC 1644 *T/TCP -TCP Extensions for Transactions Functional Specification* .

TCP Extensions for High Performance (RFC 1323)

Request For Comments 1323 suggested some TCP extensions for performance improvement over very high speed links. Two TCP options are introduced by RFC 1323: Window Scale and TCP Timestamps.

The Window Scale option allows the TCP receive window size to be larger than the present limit of 64K bytes, by defining an implicit scale factor, which is used to multiply the window size value found in the TCP header to obtain the true window size. The Window Scale option is carried in SYN segments during the connection setup phase.

If the TCP Timestamp option is enabled, the sender places a time stamp in every TCP segment, and then the receiver sends the time stamp back in the acknowledgment, allowing the sender to calculate the Round Trip Time (RTT) for each acknowledgment. This RTT estimation helps TCP to discard received duplicate segments, to calculate the retransmission timer, and to decide whether to start the slow-start process. The presence of the TCP Timestamp option also allows TCP to perform Protection Against Wrapped Sequences (PAWS). PAWS assumes that every received TCP segment contains a TCP time stamp whose values are monotonically non-decreasing in time. A segment can be discarded as an old duplicate if it is received with a time stamp less than a time stamp recently received on this connection.

The TCP Timestamp option can be enabled for a system-wide effect through the INET configuration utility 'inetcfg -set timestmp 1'. It can also be retrieved and set for each socket through the TCP_TIMESTMP socket option. Similarly, the Window Scale option can be enabled system-wide through 'inetcfg -set winscale 1,' and can also be retrieved and set for each socket through the TCP_WINSCALE socket option.

By default, the system-wide Window Scaling option is enabled and the system-wide TCP Timestamp option is disabled. Both options, if set, can be traced through the IPTRACE utility and can be displayed in the IPFORMAT dump.

For more details on these two TCP extensions, refer to RFC 1323 *TCP Extensions for High Performance* .

High Performance Send

High Performance Send is a new feature of TCP/IP 4.21 that allows an application to send data over sockets without incurring a memory copy. To use it, special memory is first allocated to the application from the TCP/IP stack with the sysctl() call. Then the application calls send(), sendto(), or sendmsg() with the MSG_MAPIO flag. The application must wait for notification from the stack that the stack has finished using the memory passed on the previous send-type call. Only then should the application reuse the memory to send more data.

There are two ways to determine if the stack has finished using the memory: event semaphores and polling. These are described in Determining if HPS Memory is Available for Reuse.

A complete example of using HPS appears in the samples directory of the toolkit.

Topics

Allocating HPS Memory
Using HPS Memory with Send Calls
Determining if HPS Memory is Available for Reuse
Freeing HPS Memory

Allocating HPS Memory

To allocate HPS memory, the application calls sysctl() as follows:

```
int mib[4];
unsigned long ptrs[15];
size_t ptrslen;
int ret;
mib[0] = CTL_OS2;
mib[1] = PF_INET;
mib[2] = IPPROTO_IP;
mib[3] = OS2_MEMMAPIO;

memset(ptr, 0, sizeof(ptrs));
ptrslen = sizeof(ptrs);
ret = sysctl(mib, sizeof(mib) / sizeof(mib[0]), ptrs,
             &ptrslen, NULL, 0);
```

The TCP/IP stack will allocate 60K bytes of memory and return 15 pointers to 4K byte blocks. These must be passed on subsequent send calls with the MSG_MAPIO flag.

To allocate memory and attach a shared-event semaphore to each block as well, the application should initialize the ptrs array with the shared-event semaphore handles before calling sysctl():

```
int mib[4];
unsigned long ptrs[15];
size_t ptrslen;
int i;
APIRET rc;
int ret;

mib[0] = CTL_OS2;
mib[1] = PF_INET;
mib[2] = IPPROTO_IP;
mib[3] = OS2_MEMMAPIO;

for (i = 0; i < sizeof(ptrs) / sizeof(ptrs[0]); i++) {
    rc = DosCreateEventSemaphore(NULL, &ptrs[i], DC_SEM_SHARED, FALSE);
    if (rc != NO_ERROR)
       exit(1);
}
ptrslen = sizeof(ptrs);
ret = sysctl(mib, sizeof(mib) / sizeof(mib[0]), ptrs, &ptrslen, NULL, 0);
```

Using HPS Memory with Send Calls

To send data using HPS, the user passes one or more of the pointers received from an HPS allocation call to one of the send-type calls along with the MSG_MAPIO flag:

```
int ret, sock;
unsigned long ptrs[15];
struct msghdr hdr;
struct iovec iovec[2];

ret = send(sock, ptrs[0], 4096, MSG_MAPIO);

hdr.msg_name = NULL;
hdr.msg_namelen = 0;
hdr.msg_iov = iovec;
hdr.msg_iovlen = 2;
```

```
 iovec[0].iov_base = ptrs[0];
 iovec[0].iov_len = 4096;
 iovec[1].iov_base = ptrs[1];
 iovec[1].iov_len = 96;
 hdr.msg_control = NULL;
 hdr.msg_controllen = 0;
 hdr.msg_flags = 0;
 ret = sendmsg(sock, &hdr, MSG_MAPIO);
```

Notes:

1. On the send() and sendto() calls, only one pointer and up to 4096 bytes may be sent per call.

2. The pointers passed to the any of the send-type calls must be exactly as returned from the allocation call; they may not be altered in any way.

Determining if HPS Memory is Available for Reuse

After a successful send-type call using HPS memory, the user must wait until the stack is finished with it before reusing it. There are two ways to determine if HPS memory is available for reuse: event semaphores and polling.

To use the event semaphores method, the application must allocate 15 shared event sempahores and pass them on the sysctl() call used to allocate the HPS memory (see the second example under Allocating HPS Memory). When the stack is finished using the HPS memory, it will post any event semaphores corresponding to the 4K blocks that are now free.

To use the polling method, the application calls sysctl() with an array of pointers. The stack will check each pointer, and if the block is in use, will zero the pointer in the array. The polling method may be used even if event semaphores were also allocated for the memory. Following is an example of the polling method:

```
long arrayofptrs[15];
int mib[4];
unsigned int needed;

mib[0] = CTL_OS2;
mib[1] = AF_INET;
mib[2] = 0;
mib[3] = OS2_QUERY_MEMMAPIO;
needed = sizeof(arrayofptrs);

if (sysctl(mib, sizeof(mib) / sizeof(mib[0]), arrayofptrs,
    &needed, NULL, 0) < 0) {
    psock_errno("sysctl(QUERY_MEMMAPIO)");
    exit(1);
}
for (i = 0; i < 15; i++) {
   if (arrayofptrs[i] == 0) {
      /* pointer is in use */
   }
}
```

Freeing HPS Memory

To free HPS memory, the application calls sysctl() as follows:

```
int mib[4];
unsigned long ptrs[15];
int ret;

mib[0] = CTL_OS2;
mib[1] = PF_INET;
mib[2] = IPPROTO_IP;
```

```
mib[3] = OS2_MEMMAPIO;

ret = sysctl(mib, sizeof(mib) / sizeof(mib[0]), NULL,
             0, ptrs, sizeof(ptrs));
```

The *ptrs* parameter is the list of array of pointers returned on the allocation call.

Passing Sockets Between Processes

Because sockets are not file handles on OS/2, sockets are not automatically passed from a parent process to a child process. Instead, sockets are global to the system and unsecure, so that any process may access any valid socket. To ensure sockets are always properly closed when a process terminates, TCPIP32.DLL installs an exit list handler (with an order code of 0x99) that closes all remaining sockets that were opened by that process. If a process attempts to pass a socket to a child process, both the parent and child need to notify TCPIP32.DLL that a change in ownership occurred so that the exit list handler for the two processes close the correct sockets when the processes terminate.

To pass ownership from parent to child, the parent process needs to issue removesocketfromlist() with the socket number that is being transferred to the child. The child process needs to issue addsockettolist() with the same socket number to assume ownership of it. After these two calls are completed, the child process's exit list handler will automatically close the socket that was passed once the child terminates (unless the child application closes the socket itself before it terminates.) See addsockettolist() and removesocketfromlist() for additional details.

Multithreading Considerations

The sockets and network utility routines are completely reentrant. Multiple threads of an application can perform any socket call.

Note: Each thread that makes sockets calls has memory automatically allocated for it by TCPIP32.DLL to store per-thread information (such as the error code for the last sockets call made on that thread). If a thread only makes protocol-independent calls, the amount of memory allocated will be small (on the order of 100 bytes). If a thread issues any of the TCP/IP network utility calls, however, a 4K block will also be allocated for that thread. None of the memory that is allocated will be deallocated until the process terminates. It will be reused however, if a thread terminates and another thread is created.

Accessing a TCP/IP API DLL from an Intermediate DLL

A TCP/IP API DLL can be accessed both directly from an application and through an intermediate DLL. An example of an intermediate DLL is a virtual network API layer that supports generalized network functionality for applications and uses the TCP/IP API.

The OS/2 Warp Toolkit contains a sample program to build a DLL. You can find the program in the SAMPLES\TCPIPTK\SAMPDLL directory.

For more information about DLLs, refer to the *OS/2 Warp Technical Library, Control Programming Guide*

Differences between OS/2 and Standard BSD Sockets

Networking services sockets is based on the Berkeley Software Distribution version 4.4 sockets implementation.

The IBM OS/2 socket implementation differs from the Berkeley socket implementation as follows:

- Sockets are not OS/2 files or devices. Socket numbers have no relationship to OS/2 file handles. Therefore, the read(), write(), and close() calls do not work for sockets: using them gives incorrect results. Use the recv(), send(), and soclose() calls instead.

- Error codes set by the OS/2 TCP/IP sockets implementation are not made available via the global *errno* variable. Instead, error codes are accessed by using the sock_errno() call (see sock_errno()). Use the psock_errno() call, instead of the perror() call, to write a short error message to the standard error device describing the last error encountered during a call to a socket library function. To access system return values, use the `errno.h` include statement supplied with the compiler.

  This is intended to obtain per-thread error codes in a multithreaded application environment and to avoid conflict with standard ANSI C error constants.

  BSD-style error checking is as follows:

  -
    ```
    rt = recv(s, buf, sizeof(buf), 0);
    if (rt == -1 && errno == EWOULDBLOCK)
    {...}
    ```

  -
    ```
    if (recv(s, buf, sizeof(buf), 0) < 0) {
    perror("Recv()");
    exit(1);
    }
    ```

  The preferred OS/2-style error checking is as follows:

  -
    ```
    rt = recv(s, buf, sizeof(buf), 0);
    if (rt == -1 && sock_errno() == SOCEWOULDBLOCK)
    {...}
    ```

  -
    ```
    if (recv(s, buf, sizeof(buf), 0) < 0)
    {
    psock_errno("Recv()");
    exit(1);
    }
    ```

  Error constants consistent with BSD sockets are provided for compatibility purposes; your application can use the error constant EWOULDBLOCK, instead of SOCEWOULDBLOCK. See Socket Error Constants, or the <NERRNO.H> file for definitions of error constants.

- Unlike the Berkeley select() call, you cannot use the OS/2 select() call to wait for activity on devices other than sockets. See select() for more information.

- The ioctl(), getsockopt(), setsockopt(), and sysctl() calls don't support all of the options supported by BSD and add some options not supported by BSD. See ioctl(), getsockopt(), setsockopt(), and sysctl() for more information.

Compiling and Linking a Sockets API Application

Follow these steps to compile and link a sockets API application using the IBM VisualAge C++ compiler:

1. To compile your program, enter:

   ```
   icc /Gm /c myprog.c
   ```

2. To create an executable program, you can enter:

   For VisualAge C++

```
ilink /NOFREEFORMAT myprog,myprog.exe /STACK:0x4000
```

Note: For more information about the compile and link options, multithreaded libraries, and dynamic link libraries, refer to the User's Guide provided with your compiler.

Sample Programs

The following sample programs are included in the TCP/IP toolkit:

SOCKET	These samples consist of several C socket client-server programs.
SAMPDLL	These samples demonstrate building and using a DLL that uses TCP/IP.
HPS	These samples demonstrate using High Performance Send.
RPC	These samples provide examples of Remote Procedure Call (RPC) client, server, and raw data stream programs.
RPCGEN	The samples define remote procedure characteristics and demonstrate an RPC client and server program.

Sockets in the Internet Domain

This section describes the use of sockets in the internet domain.

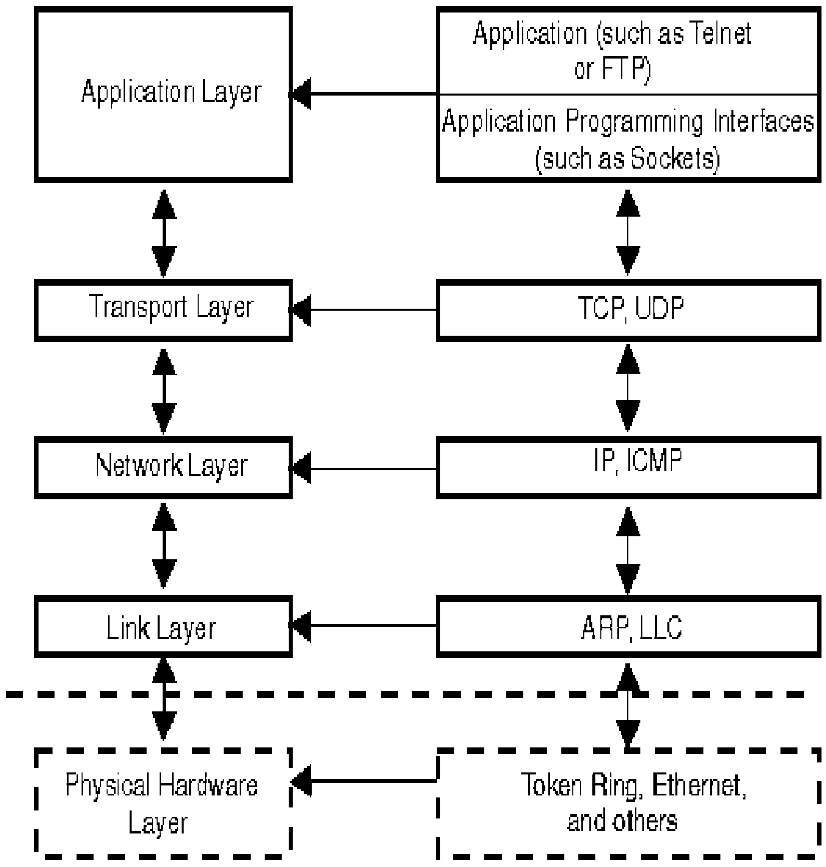
Topics

Protocols Used in the Internet Domain
Getting Started with Sockets in the Internet Domain
Network-Byte Order
Internet Address Formats
TCP/IP-Specific Network Utility Routines
The _res Data Structure
Ports
IP Multicasting
Socket Secure Support

Protocols Used in the Internet Domain

This section describes the network protocols in TCP/IP. The internet domain is supported only by the TCP/IP protocol suite. Networking protocols like TCP/IP are layered as shown in the following figure. For more information on the internet domain and the TCP/IP protocol suite, refer to *TCP/IP Illustrated, Volume 1: The Protocols,* W. Richard Stevens, Addison-Wesley Publishing Co., 1994.

The Internet Layered Architecture


Application Layer
Application (such as Telnet or FTP)
Application Programming Interfaces (such as Sockets)
Transport Layer
TCP, UDP
Network Layer
IP, ICMP
Link Layer
ARP, LLC
Physical Hardware Layer
Token Ring, Ethernet, and others

Topics

Transmission Control Protocol (TCP)
User Datagram Protocol (UDP)
Internet Protocol (IP)
Internet Control Message Protocol (ICMP)
Address Resolution Protocol (ARP)
Internet Group Management Protocol (IGMP)

Transmission Control Protocol (TCP)

TCP is a transport protocol that provides a reliable mechanism for delivering packets between hosts on an internet. TCP takes a stream of data, breaks it into datagrams, sends each one individually using Internet Protocol (IP), and reassembles the datagrams at the destination node. If any datagrams are lost or damaged during transmission, TCP detects this and resends the missing datagrams. The received data stream is a reliable copy of the transmitted data stream.

Note: The PUSH flag is a notification from the sender to the receiver for the receiver to pass all the data that it has to the receiving process. This data consists of whatever is in the segment with the PUSH flag, along with any other data the receiving TCP has collected for the receiving process. Our TCP implementation automatically sets the PUSH flag if the data in the segment being sent empties the send buffer. In addition, our implementation ignores a received PUSH flag because we don't delay the delivery of the received data to the application.

You can use TCP sockets for both passive (server) and active (client) applications. While some calls are necessary for both types, some are role-specific. TCP is the default protocol for stream sockets in the internet domain. For sample C socket communication client and server programs, see the TCP/IP samples included in the OS/2 Warp Toolkit.

TCP is a connection-oriented protocol. It is used to communicate between pairs of applications. After you make a connection, it exists until you close the socket. During the connection, data is either delivered or an error code is returned by networking services.

User Datagram Protocol (UDP)

UDP is a transport-layer datagram protocol that sends and receives whole packets across the network. UDP is used for application-to-application programs between TCP/IP hosts. UDP does not offer a guarantee of datagram delivery or duplication protection. UDP does provide checksums for both the header and data portions of a datagram. However, applications that require reliable delivery of streams of data should use TCP. UDP is the default protocol for datagram sockets in the internet domain.

Unlike applications using TCP, UDP applications are usually connectionless. A UDP socket application can become connected by calling the connect() API. An unconnected socket can be used to communicate with many hosts; but a connected socket, because it has a dedicated destination, can exchange data with only one host at a time.

UDP is considered an unreliable protocol because it sends its data over the network without verification. Consequently, after a packet has been accepted by the UDP interface, neither the arrival of the packet nor the arrival order of the packet at the destination can be guaranteed.

Internet Protocol (IP)

The IP network layer provides the interface from the transport layer (host-to-host) protocols to the link-level protocols. IP is the basic transport mechanism for routing IP packets to the next gateway, router, or destination host.

IP provides the means to transmit packets of data from sources to destinations. Sources and destinations are hosts identified by 32-bit IP addresses, which are assigned independent of the underlying physical network. Outgoing packets automatically have an IP header prepended to them, and incoming packets have their IP header removed before being passed to the higher-level protocols. This protocol ensures the unique addressing of hosts in an internet network.

IP does not ensure a reliable communication, because it does not require acknowledgments from the sending host, the receiving host, or intermediate hosts. IP does not provide error control for data; it provides only a header checksum. IP treats each packet as an independent entity, unrelated to any other packet. IP does not perform retransmissions or flow control. A higher-level protocol like TCP (Transmission Control Protocol) that uses IP must implement its own reliability procedures.

Applications do not typically access IP directly, but rather use TCP or UDP which, in turn, use IP. Raw sockets can use IP.

Internet Control Message Protocol (ICMP)

ICMP is used to pass control information between hosts. For example, the information can be sent in any of the following situations:

- When a host checks to see if another host is available (PING)
- When a packet cannot reach its destination
- When a gateway or router can direct a host to send traffic on a shorter route
- When a gateway or router does not have the buffering capacity to forward a packet

ICMP provides feedback about problems in the communication environment; it does not make IP reliable. The use of ICMP does not guarantee that an IP packet will be delivered reliably or that an ICMP message will be returned to the source host when an IP packet is not delivered or is incorrectly delivered.

Raw sockets can use ICMP and, like IP, ICMP is not typically used by application programs directly.

Address Resolution Protocol (ARP)

ARP maps IP addresses to hardware addresses. TCP/IP uses ARP to collect and distribute the information for mapping tables.

ARP is not directly available to users or applications. When an application sends an internet packet, IP requests the appropriate address mapping. If the mapping is not in the mapping table, an ARP broadcast packet is sent to all the hosts on the local network requesting the physical hardware address for the host.

Proxy ARP allows an assigned substitute ARP agent (typically a router) to respond to ARP requests on behalf of certain hosts which reside on the outside of a network. A proxy ARP agent must be defined beforehand for the ARP hosts which will have their -P (Public) flag set.

Internet Group Management Protocol (IGMP)

RFC 1112 defines IGMP, which describes interactions between IP multicast hosts and multicast routers. A multicast router needs to know the current membership of host groups in all attached local networks in order to forward multicast datagrams to hosts on local networks. IP multicast datagrams will not be forwarded to local networks if there are no members of the destination host group.

There are two types of IGMP messages transmitted on local networks. Both types of messages are transmitted by multicasting to reduce network load. A multicast router periodically sends IGMP membership queries to hosts on the same network. An IGMP membership query is sent to the all-hosts group (224.0.0.1).

Upon receiving a membership query from a multicast router, a multicast host starts a random timer for each host group joined on the interface that receives IGMP membership queries. A host sends IGMP membership reports when timers expire. Membership reports are sent to the host group being reported.

If other hosts on the same network are to receive an IGMP membership report on the same host group, these hosts should cancel the timer before it expires. This prevents duplicated IGMP membership reports from flooding a local network. An IGMP membership report is also sent when a host joins a new host group.

When a multicast router receives an IGMP membership report of one host group, the router updates its knowledge of the current membership on a particular network. If no reports are received on a particular host group after several queries, a multicast router assumes that there are no local members on that host group and stops forwarding any multicast datagrams with that destination host group.

Getting Started with Sockets in the Internet Domain

This section provides some basic information for getting started with sockets in the internet domain:

- Use the socket() call to create a socket in the internet domain specifying PF_INET for the *domain* parameter.
- Use AF_INET for the address family.
- The following socket types are supported for the internet domain:
  - Datagram (SOCK_DGRAM)
  - Raw (SOCK_RAW)
  - Stream (SOCK_STREAM)

  The socket type is passed as a parameter to the *socket()* call. For additional information, see Socket Types and general socket programming concepts in Sockets General Programming Information.
- Datagram sockets use the UDP protocol, stream sockets use the TCP protocol, and raw sockets can use the raw, ICMP, or IGMP protocols.
- Use the network utility routines to get addresses with a given name (see TCP/IP-Specific Network Utility Routines for additional information).

Network-Byte Order

Ports and addresses are specified to sockets API calls by using the network-byte ordering convention. Network-byte order is also known as *big endian* byte ordering, which has the high-order byte at the starting address. By contrast, *little endian* has the low-order byte at the starting address. Using network-byte ordering for data exchanged between hosts allows hosts using different underlying byte ordering conventions to exchange address information. There is a set of network utility functions for translating addresses from host-byte to network-byte order and from network-byte to host-byte order. For more information about network-byte order and address translation, see:

- bind()
- htonl()
- htons()
- ntohl()
- ntohs()

Note: The socket interface does not handle application data byte ordering differences. Application writers must handle data buffer byte order differences themselves.

Internet Address Formats

This section describes the address formats used in the internet domain.

internet addresses (IP) are 32-bit values that represent a network interface. Every internet address within an administered internet (AF_INET) communication domain must be unique. A host can have as many internet addresses as it has network interfaces. For more information about internet address formats, see *Internetworking with TCP/IP Volume I: Principles, Protocols, and Architectures* , and *Volume II: Implementation and Internals* , Douglas E. Comer, Prentice Hall, 1991.

Each internet host is assigned at least one unique internet address. This address is used by IP and other higher-level protocols. When a host is a gateway, it has more than one IP address. Gateway hosts connect two or more physical networks and have one IP address per connected physical network.

Addresses within an internet consist of a network number and a local address. All physical host IP addresses share the same network number and are logically part of the same network even if that network is connected with various physical media.

Hosts on disjoint physical networks might also have the same network number, but are not part of the same internet network. Hosts that are part of the same internet network can exchange packets directly without going through intermediate routers. An internet network can be subdivided logically using a subnet mask. All host interfaces to the same physical network are given the same subnetwork number. An internet domain can provide standards for assigning addresses to networks, broadcasts, and subnetworks.

Dotted-Decimal Notation: A commonly used notation for internet host addresses is the dotted-decimal format, which divides the 32-bit address into four 8-bit fields. The value of each field is specified as a decimal number, and the fields are separated by periods (for example, 10.2.0.52).

Address examples in this document use dotted-decimal notation in the following forms:

- *nnn.lll.lll.lll*
- *nnn.nnn.lll.lll*
- *nnn.nnn.nnn.lll*

where:

nnn	represents part or all of a network number.
lll	represents part or all of a local address.

Note: Additional details about internet network address format class A, B, C, and D addresses, subnetwork address format, and broadcast address formats can be found in the *TCP/IP Guide*

Addressing within an Internet Domain: A socket address in an internet communication domain is composed of five fields in the following sockaddr_in structure: length, address family, port, internet address, and a .* reserved field. The sockaddr_in structure should be cleared before use. The structure is located in the <NETINET\IN.H> header file:

```
struct in_addr {
        u_long s_addr;
};




struct sockaddr_in {
        u_char sin_len;             /* sizeof (struct sockaddr_in) = 16 */
        u_char sin_family;          /* AFINET */
        u_short sin_port;           /* port id */
        struct  in_addr sin_addr;  /* address */
        char    sin_zero[8];       /* not used */
};
```

The *sin_len* field is set to 16 as the size of the sockaddr_in structure.

The *sin_family* field is set to AF_INET.

The *sin_port* field is set to the port number in network-byte order. If you are specifying your workstation address in *sin_addr* and you set *sin_port* to 0 using the bind() call, the system assigns an available port. If you specify a different workstation address in *sin_addr*, you must specify the port. For more information on ports, see Ports.

The *sin_addr* field is set to the internet address represented in network-byte order. When specified as a parameter to bind(), *sin_addr* is usually set to the constant INADDR_ANY, as defined in <NETINET\IN.H>. This binds the socket to any and all local internet addresses. By using INADDR_ANY, an application can bind a socket without specifying the local internet address. The constant INADDR_ANY also allows an application running on a host with multiple interfaces (called a multihomed host) to receive UDP datagrams and TCP connection requests arriving at any interface on a single socket. (The application is not required to have one socket per interface, with each interface bound to a specific internet address).

To specify your workstation address, you can leave *sin_addr* unspecified. If you are specifying a different workstation address, you must specify a valid internet address for that workstation.

The *sin_zero* field is not used, and it should be set to 0 by the application before passing the address structure to any sockets call.

TCP/IP-Specific Network Utility Routines

This section describes the library of network utility routines.

Network utility routines are described in the following sections.

Topics

Host Names Information
Network Names Information
Protocol Names Information
Service Names Information
Network Address Translation
Network-Byte Order Translation
Internet Address Manipulation
Domain Name Resolution

Host Names Information

The following is a list of host-related calls:

- gethostbyname()
- gethostbyaddr()
- sethostent()
- gethostent()
- endhostent()
- gethostname()
- gethostid()

The gethostbyname() call takes an internet host name and returns a hostent structure, which contains the name of the host, aliases, host address family and host address. The hostent structure is defined in the <NETDB.H> header file. The gethostbyaddr() call maps the internet host address into a hostent structure.

The database for these calls is provided by the name server or the ETC\HOSTS file if a name server is not present or is unable to resolve the host name.

The sethostent(), gethostent(), and endhostent() calls open, provide sequential access to, and close the ETC\HOSTS file.

The gethostname() call gets the name for the local host machine.

The gethostid() call returns an integer that identifies the host machine. Host IDs fall under the category of internet network addressing because, by convention, the 32-bit internet address is used.

Network Names Information

The following is a list of network-related calls:

- getnetbyname()
- getnetbyaddr()
- setnetent()
- getnetent()
- endnetent()

The getnetbyname() call takes a network name and returns a netent structure, which contains the name of the network, aliases, network address family, and network number. The netent structure is defined in the <NETDB.H> header file. The getnetbyaddr() call maps the network number into a netent structure.

The database for these calls is provided by the ETC\NETWORKS file.

The setnetent(), getnetent(), and endnetent() calls open, provide sequential access to, and close the ETC\NETWORKS file.

Protocol Names Information

The following is a list of protocol related calls:

- getprotobyname()
- getprotobynumber()
- setprotoent()
- getprotoent()
- endprotoent()

The getprotobyname() call takes the protocol name and returns a protoent structure, which contains the name of the protocol, aliases, and protocol number. The protoent structure is defined in the <NETDB.H> header file. The getprotobynumber() call maps the protocol number into a protoent structure.

The database for these calls is provided by the ETC\PROTOCOL file.

The setprotoent(), getprotoent(), and endprotoent() calls open, provide sequential access to, and close the ETC\PROTOCOL file.

Service Names Information

The following is a list of service related calls:

- getservbyname()
- getservbyport()
- setservent()
- getservent()
- endservent()

The getservbyname() call takes the service name and protocol, and returns a servent structure that contains the name of the service, aliases, port number, and protocol. The servent structure is defined in the <NETDB.H> header file. The getservbyport() call maps the port number and protocol into a servent structure.

The database for these calls is provided by the ETC\SERVICES file.

The setservent(), getservent(), and endservent() calls open, provide sequential access to, and close the ETC\SERVICES file.

Network Address Translation

Network library calls enable an application program to locate and construct network addresses while using interprocess communication facilities in a distributed environment.

Locating a service on a remote host requires many levels of mapping before client and server can communicate. A network service is assigned a name that is intended to be understandable for a user. This name and the name of the peer host must then be translated into network addresses. Finally, the address must then be used to determine a physical location and route to the service.

Network library calls map:

- Host names to network addresses
- Network names to network numbers
- Protocol names to protocol numbers
- Service names to port numbers

Additional network library calls exist to simplify the manipulation of names and addresses.

An application program must include the <NETDB.H> file when using any of the network library calls.

Note: All networking services return values in standard network byte order.

Network-Byte Order Translation

Internet domain ports and addresses are usually specified to calls using the network-byte ordering convention. The following calls translate integers from host- to network-byte order and from network- to host-byte order.

htonl()	Translates host to network, long integer (32-bit)
htons()	Translates host to network, short integer (16-bit)
ntohl()	Translates network to host, long integer (32-bit)
ntohs()	Translates network to host, short integer (16-bit)

Internet Address Manipulation

The following calls convert internet addresses and decimal notation, and manipulate the network number and local network address portions of an internet address:

inet_addr()	Translates dotted-decimal notation to a 32-bit internet address (network-byte order).
inet_network()	Translates dotted-decimal notation to a network number (host-byte order), and zeros in the host part.
inet_ntoa()	Translates 32-bit internet address (network-byte order) to dotted-decimal notation.
inet_netof()	Extracts network number (host-byte order) from 32-bit internet address (network-byte order).
inet_lnaof()	Extracts local network address (host-byte order) from 32-bit internet address (network-byte order).
inet_makeaddr()	Constructs internet address (network-byte order) from network number and local network

address.

Domain Name Resolution

In TCP/IP, communication is based on internet addresses. When a TCP/IP application receives a symbolic host name, it calls a host name resolver routine to resolve the symbolic name into an internet address. The host name resolver routine queries a domain name server or a local HOSTS file, or both, to perform the name resolution.

If a RESOLV2 file or RESOLV file exists in the ETC subdirectory, the host name resolver routine first tries to resolve the name by querying the name servers specified in that file.

If resolution through a name server fails or if a RESOLV2 or RESOLV file does not exist, the host name resolver routine tries to resolve the name locally by searching the HOSTS file in the ETC subdirectory for a match of the symbolic host name.

The above search order can be reversed by setting the OS/2 environment variable USE_HOST_FIRST to any nonzero value. If this variable is set, the host name resolver routine searches the local HOSTS file first before querying a domain name server.

If a match is found, the routine returns the corresponding internet address. If a match is not found, the routine displays a message stating that the host is unknown.

RESOLV and RESOLV2 files

NETWORK CONNECTION	RESOLV FILE	RESOLV2 FILE
LAN only connection		X
SLIP only connection	X	
LAN and SLIP connections	X	X (used for domain name resolution)

The following resolver calls are used to make, send, and interpret packets for name servers in the internet domain:

- res_query()
- res_querydomain()
- res_mkquery()
- res_send()
- res_search()
- res_init()
- dn_comp()
- dn_expand()
- dn_find()
- dn_skipname()
- _getshort()
- _getlong()
- putlong()
- putshort()

The _res Data Structure

Global information used by these resolver calls is kept in the _res data structure. This structure is defined in the <RESOLV.H> file and contains the following members:

Type	Member	Contents

`int`	`retrans`	`Retransmission time interval`
`int`	`retry`	`Number of times to retransmit`
`long`	`options`	`Option flags`
`int`	`nscount`	`Number of name servers`
`struct sockaddr_in[MAXNS]`	`nsaddr_list`	`Address of name server`
`unsigned short`	`id`	`Current packet id`
`char[MAXDNAME]`	`defdname`	`Default domain`
`char*[MAXDNSRCH+1]`	`dnsrch`	`Components of domain to search`

The options field of the _res data structure is constructed by logically ORing the following values:

RES_INIT
: Indicates whether the initial name server and default domain name have been initialized (that is, whether the res_init() call has been called).

RES_DEBUG
: Prints debugging messages.

RES_USEVC
: Uses Transmission Control Protocol/Internet Protocol (TCP/IP) connections for queries instead of User Datagram Protocol/Internet Protocol (UDP/IP).

RES_STAYOPEN
: Used with the RES_USEVC value, keeps the TCP/IP connection open between queries. While UDP/IP is the mode normally used, TCP/IP mode and this option are useful for programs that regularly perform many queries.

RES_RECURSE
: Sets the Recursion Desired bit for queries. This is the default.

RES_DEFNAMES
: Appends the default domain name to single-label queries. This is the default.

These environment variables affect values related to the _res data structure:

LOCALDOMAIN
: Overrides the default local domain, which is read from the ETC\RESOLV.conf file and stored in the defdname field of the _res data structure.

RES_TIMEOUT
: Overrides the default value of the retrans field of the _res data structure, which is the value of the RES_TIMEOUT constant defined in the <RESOLV.H> file. This value is the base timeout period in seconds between queries to the name servers. After each failed attempt, the timeout period is doubled. The timeout period is divided by the number of name servers defined. The minimum timeout period is 1 second.

RES_RETRY
: Overrides the default value for the retry field of the _res data structure, which is 4. This value is the number of times the resolver tries to query the name servers before giving up. Setting RES_RETRY to 0 prevents the resolver from querying the name servers.

The res_send() call does not perform interactive queries and expects the name server to handle recursion.

Ports

A port is used to differentiate between multiple applications on a host using the same protocol (TCP or UDP). It is an additional qualifier used by the system software to get data to the correct application. Physically, a port is a 16-bit integer. Some ports are reserved for particular applications and are called *well-known ports* . Well-Known Port Assignments contains the well-known port assignments list.

IP Multicasting

This section presents general concepts as well as technical implementation details about IP multicasting.

Multicasting enables a message to be transmitted to a group of hosts, instead of having to address and send the message to each group member individually. This reduces the traffic impact on networks with, for example, audio/video applications involving more than two participants. internet addressing provides for Class D addressing that is used for multicasting.

Topics

Multicasting and the setsockopt() Call
A Socket Program Example

Multicasting and the setsockopt() Call

When a datagram socket is defined, the setsockopt() call can be used to modify it. In order to join or leave a multicast group, use the setsockopt() call with the IP_ADD_MEMBERSHIP or IP_DROP_MEMBERSHIP flags. The interface that is used and the group used are specified in an ip_mreq structure that contains the following fields:

```
struct ip_mreq{
   struct in_addr imr_multiaddr;
   struct in_addr imr_interface;
}
```

The in_addr structure is defined as:

```
struct in_addr{
   ulong s_addr;
}
```

A *host group* consists of zero or more IP hosts. An IP datagram designated to a host group address will be delivered to all the current members of that group. A host group does not have a fixed membership. Any IP multicast-capable hosts can join or leave a host group dynamically.

To join or leave a host group, an application needs to specify a host group address, ranging from 224.0.0.2 to 239.255.255.255, and a network interface address. (Note that 224.0.0.0 is reserved and 224.0.0.1 is permanently assigned to the *all hosts group,* which includes all hosts and routers participating in IP multicast.) It is possible to join the same host group on more than one network interface. It is also possible for more than one application to join the same host group. A host's IP module discards an incoming multicast datagram designated for a host group not joined on that incoming network interface, even though the same host group is joined on another network interface.

In order to send to a multicasting group it is not necessary to join the groups. For receiving transmissions sent to a multicasting group membership is required. For multicast sending use an IP_MULTICAST_IF flag with the setsockopt() call. This specifies the interface to be used.

An application can specify the time-to-live value of outgoing multicast datagrams. The default value is one for all IP multicast datagrams. An application can also specify whether a copy of the multicast datagram is looped back in the case where the host itself is a member of the destination host group. By default, a copy of the multicast datagram is looped back.

It may be necessary to call the setsockopt() call with the IP_MULTICAST_LOOP flag in order to control the loopback of multicast packets. By default, packets are delivered to all members of the multicast group including the sender, if it is a member. However, this can be disabled with the setsockopt() call using the IP_MULTICAST_LOOP flag.

For hosts that attach to more than one network, an application can choose which interface is used for the initial transmission. Only one interface can be used for the initial transmission. Further transmission to other networks is the responsibility of *multicast routers.* Do not be confused by the interface where a host group joins the outgoing interface for multicast transmission. The interface specified when joining or leaving a host group is mainly for receiving incoming multicast datagrams. An application can join one host group on one network interface but transmit data to the same host group by way of another interface.

Currently multicast is supported only on UDP, so datagram sockets should be used to do multicast operations. One thing to consider is that using aliasing and multicasting together (with multiple processes) may give unexpected results, and the following limitations apply.

- More than one socket can bind on Class D IP address (or mcast address) and a common port; for example, two clients that want to receive the same mcast packet.
- These sockets must also set a socket option, SO_REUSEADDR.

The setsockopt() call flags that are required for multicast communication and used with the IPPROTO_IP protocol level follow:

IP_ADD_MEMBERSHIP
: Joins a multicast group as specified in the *optval* parameter of type struct ip_mreq. A maximum of 20 groups may be joined per socket.

IP_DROP_MEMBERSHIP
: Leaves a multicast group as specified in the *optval* parameter of type struct ip_mreq. Only allowable for processes with UID=0.

IP_MULTICAST_IF
: Permits sending of multicast messages on an interface as specified in the *optval* parameter of type struct in_addr. An address of INADDR_ANY (0x000000000) removes the previous selection of an interface in the multicast options. If no interface is specified then the interface leading to the default route is used.

IP_MULTICAST_LOOP
: Sets multicast loopback, determining whether or not transmitted messages are delivered to the sending host. An *optval* parameter of type uchar is used to control loopback being on or off.

IP_MULTICAST_TTL
: Sets the time-to-live for multicast packets. An *optval* parameter of type uchar is used to set this value between 0 and 255.

The getsockopt() function can also be used with the multicast flags to obtain information about a particular socket:

IP_MULTICAST_IF
: Retrieves the interface's IP address.

IP_MULTICAST_LOOP
: Retrieves the specified looping mode from the multicast options.

IP_MULTICAST_TTL
: Retrieves the time-to-live in the multicast options.

The following examples demonstrate the use of the setsockopt() call with the protocol level set to the Internet Protocol (IPPROTO_IP).

To mark a socket for sending to a multicast group on a particular interface:

```
struct ip_mreq imr;
setsockopt(s, IPPROTO_IP, IP_MULTICAST_IF, &imr.imr_interface,
sizeof(struct
in_addr));
```

To disable the loopback on a socket:

```
char loop = 0;
setsockopt(s, IPPROTO_IP, IP_MULTICAST_LOOP, &loop, sizeof(uchar));
```

To allow address reuse for binding multiple multicast applications to the same IP group address:

```
int on = 1;
setsockopt(s, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(int));
```

To join a multicast group for receiving:

```
struct ip_mreq imr;
setsockopt(s, IPPROTO_IP, IP_ADD_MEMBERSHIP, &imr, sizeof(struct
ip_mreq));
```

To leave a multicast group:

```
struct ip_mreq imr;
setsockopt(s, IPPROTO_IP, IP_DROP_MEMBERSHIP, &imr, sizeof(struct
ip_mreq));
```

A Socket Program Example

Following is an example of a socket application using IP multicasting.

```
struct sockaddr_in *to = (struct sockaddr_in *)&group;  /* group address     */
struct sockaddr_in listen_addr;
struct ip_mreq imr;                          /* multicast request structure  */
struct in_addr ifaddr;                       /* multicast outgoing interface */
short  loop = 0;                             /* don't loop back              */
short  ttl = 16;                             /* multicast time-to-live       */

       sock_init();

       /* init */
       imr.imr_multiaddr.s_addr = 0xe0010101;     /* 224.1.1.1 */
       imr.imr_multiaddr.s_addr = htonl(imr.imr_multiaddr.s_addr);
       imr.imr_interface.s_addr = INADDR_ANY;
       imr.imr_interface.s_addr = htonl(imr.imr_interface.s_addr);
       ifaddr.s_addr = INADDR_ANY;
       ifaddr.s_addr = htonl(ifaddr.s_addr);

       bzero( (char *)&group, sizeof(struct sockaddr_in) );
       to->sin_len = sizeof(to);
       to->sin_family = AF_INET;
       to->sin_port = 1201;                    /* some port number */
       to->sin_port = htons(to->sin_port);
       to->sin_addr.s_addr = imr.imr_multiaddr.s_addr;
       listen_addr = (*to);

       sock = socket(AF_INET, SOCK_DGRAM, 0);

       if (sock <= 0)  {
       psock_errno("Bad socket");
       exit(1);
       }

       /* join group */
       if( setsockopt( sock, IPPROTO_IP, IP_ADD_MEMBERSHIP,
       &imr, sizeof(struct ip_mreq) ) == -1 )  {
       psock_errno( "can't join group" );
       exit(1);
       }

       /* set multicast options */
       if (setsockopt(sock, IPPROTO_IP, IP_MULTICAST_IF,
       &ifaddr, sizeof(ifaddr)) == -1) {
       perror ("can't set multicast source interface");
       exit(1);
       }
       if (setsockopt(sock, IPPROTO_IP, IP_MULTICAST_LOOP, &loop, 1) == -1)) {
       psock_errno ("can't disable multicast loopback");
       exit(1);
       }
       if (setsockopt(sock, IPPROTO_IP, IP_MULTICAST_TTL, &ttl, 1) == -1)) {
       psock_errno ("can't set multicast ttl");
       exit(1);
       }

       /* bind to a group address */
       if (bind(sock, &listen_addr, sizeof(struct sockaddr_in)) != 0) {
       psock_errno("Binding multicast socket");
       exit(1);
       }

       .
       .
       .

       /* leave group */
       if( setsockopt( sock, IPPROTO_IP, IP_DROP_MEMBERSHIP,
       &imr, sizeof(struct ip_mreq) ) == -1 )  {
       psock_errno( "can't leave group" );
       exit(1);
       }
       .
       .
       .
```

Socket Secure Support

OS/2 supports Socket Secure Support (SOCKS) Version 4. This section presents information about OS/2 SOCKSification.

Topics

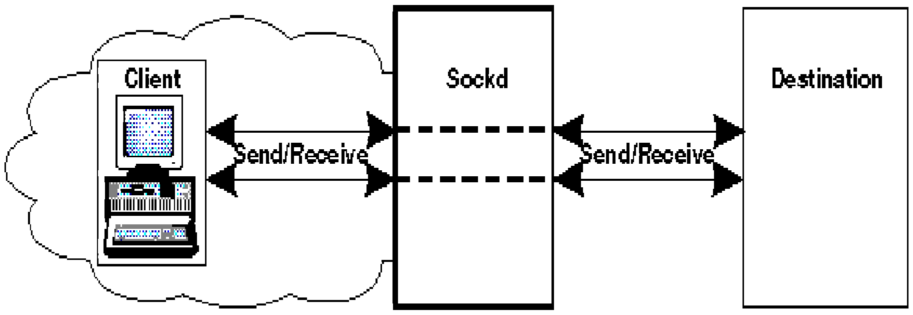
The SOCKS Server
The SOCKS Library

The SOCKS Server

A SOCKS server is a type of firewall that protects computers in a network from access by users outside that network. A SOCKS server is similar to a proxy gateway in that they both work as proxy agents but their approaches are different.

OS/2 SOCKS support for TCP/IP applications allows client applications to interact with a SOCKS server. OS/2 SOCKSified DLLs get the necessary information from socks.cfg. The DLL checks to determine if the connection should go through SOCKS. If socks_flag is on in socks.env and socks.cfg indicates that the connection should go through socks, then the connection goes through SOCKS. Otherwise, the connection does not go through SOCKS. The following figure illustrates how a typical write() to a socket would appear.

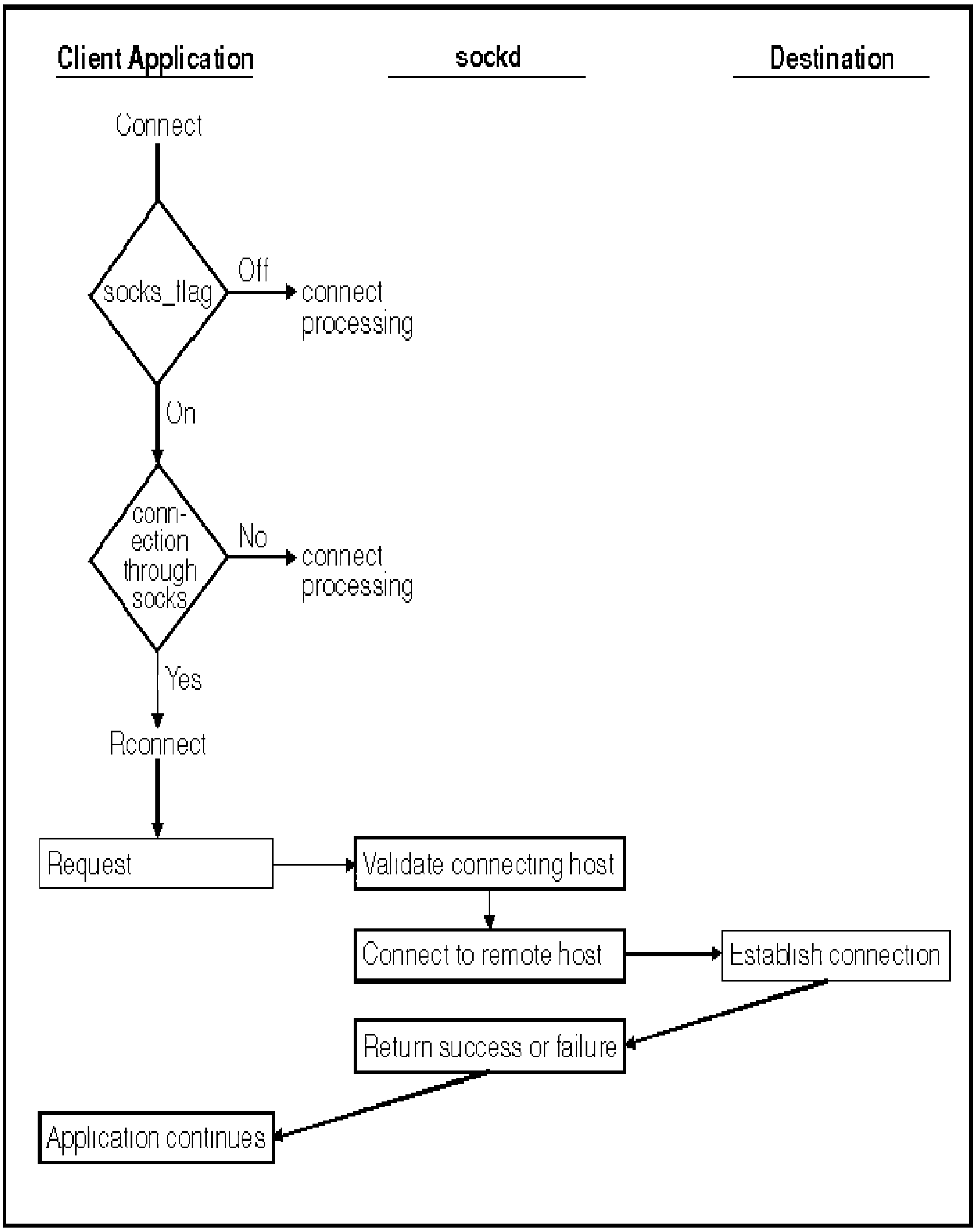
A Typical Write() to a SOCKS Server



With SOCKS support, client applications connect to the SOCKS server and then the SOCKS server connects to an external network. The SOCKS server verifies that a host name and user ID are allowed to access an internet.

If you are using a SOCKS server, connect() will call Rconnect(). See connect() for information about the connect() call. The following figure illustrates connect().

connect() Request



From the user's point of view (behind the firewall host within the local area network), there is no difference between running SOCKS and the regular client software on a host. All connections at the application level appear to work the same, with the hidden difference that all traffic is passing through **sockd** on the firewall host. This transparency is achieved through the Socks library routines.

Connect requests are originated by a call to Rconnect() on the internal host. This causes **sockd** to establish a connection to the remote host and return a success or fail response. At this point, the application can read and write to the socket connection to the firewall and **sockd** acts as a bridge between the local and external socket connections.

The SOCKS Library

The SOCKS library calls establish connections to **sockd** on the firewall and transmit information. The **sockd** daemon performs the operation as if it were originating the request. Any data **sockd** receives from the external connection is then passed on to the original requestor. To the internal host everything appears as usual, but to the external host **sockd** appears to be the originator of the communication.

The SOCKS library routines pass all network connections to **sockd,** which is running on the firewall. The functions that are provided are designated by the letter R preceding the name of the regular C library socket calls that they are replacing. The SOCKS routines take the same parameters as the original functions, with the exception of Rbind(). Rbind() has an additional parameter: the address of the remote host from which the connection is established.

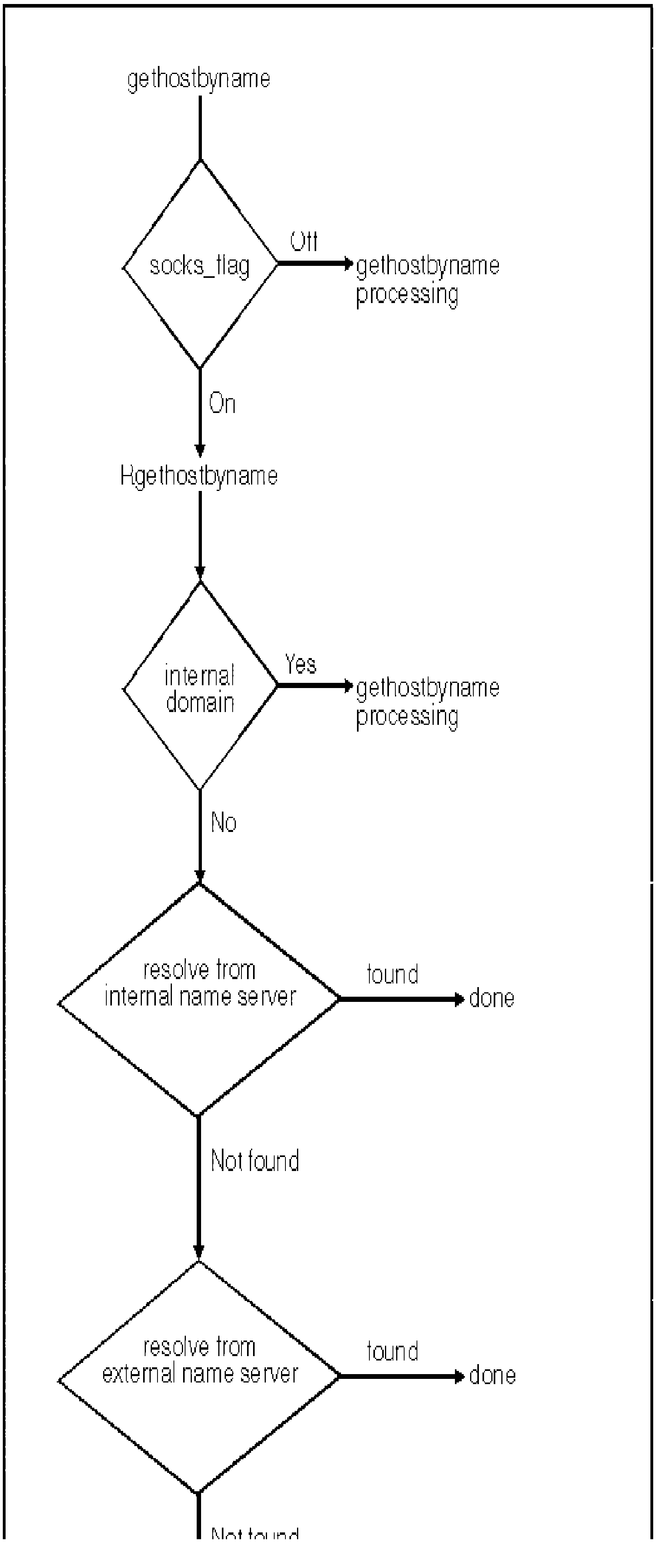
The following routines are supported:

Routine	**Parameters**
Raccept	(int socket, struct sockaddr*addr, int addrlen)
Rbind	(int socket, struct sockaddr*name, int namelen, *struct sockaddr*remote)*
Rconnect	(int socket, struct sockaddr*name, int namelen)
Rgethostbyname	(char *host)
Rgetsockname	(int socket, struct sockaddr*name, int namelen)
Rlisten	(int socket, int backlog)

See the information for the regular calls in Protocol-Independent C Sockets API.

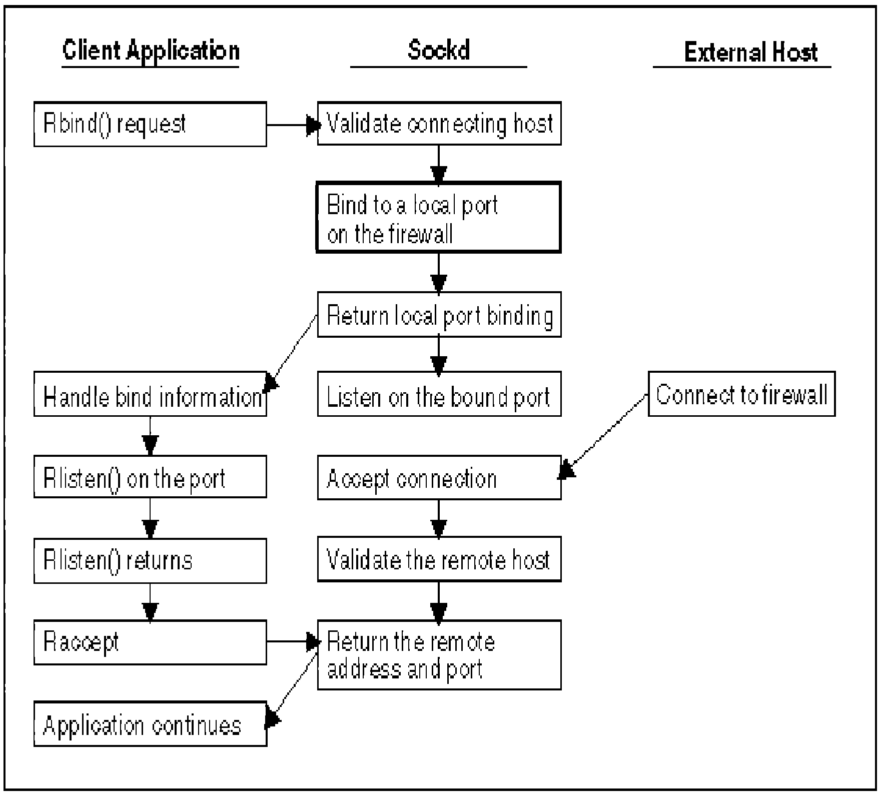
The following figure shows how a gethostbyname() call works.

gethostbyname


gethostbyname
socks_flag
Off
gethostbyname processing
On
Rgethostbyname
internal domain
Yes
gethostbyname processing
No
resolve from internal name server
found
done
Not found
resolve from external name server
found
done
Not found

The following figure shows how an Rbind() request works.

Rbind() Request



Sockets over Local IPC

This section describes how sockets over Local Inter-Processor Communication (IPC) allow the programmer to communicate between applications on the same machine using the sockets API. Local IPC sockets are not bound to a network protocol but rather use the underlying host facilities to provide high-performance IPC.

Topics

Getting Started with Sockets Over Local IPC
Local IPC Address Format

Getting Started with Sockets Over Local IPC

This section provides some basic information for getting started with sockets over Local IPC:

- Use PF_OS2, or PF_UNIX for the protocol family
- Use AF_OS2, or AF_UNIX for the address family
- The following socket types are supported for the Local IPC domain:
  - Datagram (SOCK_DGRAM)
  - Stream (SOCK_STREAM)

The socket type is passed as a parameter to the socket() call. For additional information, see Socket Types.

- A unique text string is used as a name. See Local IPC Address Format for details.

- If a connect() socket call is received without an explicit bind() call, an implicit bind is automatically performed. The application can use any name, and a unique Local IPC name is generated by networking services. You can retrieve the Local IPC name by using the getsockname() call.

Local IPC Address Format

A socket address in a local system is composed of three fields in the following sockaddr_un structure: length, address family, and path name. The structure is located in the <SYS\UN.H> header file:

```
struct sockaddr_un {
    u_char  sun_len;             /* sockaddr len including null */
    u_char  sun_family;          /* AF_OS2 or AF_UNIX */
    char    sun_path[108];       /* path name */
};
struct sockaddr_un un;
```

The *sun_family* field is set to AF_OS2 or AF_UNIX.

The *sun_path* field is the OS/2 Warp file and path name to be used as the address of the Local IPC socket. If the address is NULL, the bind call binds a unique local address to the socket descriptor *s*. Each address is a combination of address family (*sun_family*) and a character string (*sun_path*) no longer than 108 characters.

Each socket must use a unique character string as its local name to bind a name to a socket. The name in *sun_path* should begin with "\socket\".

For example,

```
struct sockaddr_un un;
int sd;
sd = socket(PF_OS2, SOCK_STREAM, 0);
memset(&un, 0, sizeof(un));

un.sun_len = sizeof(un);
un.sun_family = AF_OS2;
strcpy(un.sun_path, "\socket\XYZ", 12);
bind(sd, (struct sockaddr *)&un, sizeof(un));
```

Sockets over NetBIOS

This section describes the use of sockets with NetBIOS. Each application assigns itself one or more NetBIOS names for each adapter. The NetBIOS protocol maintains a table of the names that a node is known by on the network. NetBIOS supports two types of names: *unique* and *group.* When the name is unique, the application binds the name and NetBIOS checks the network to ensure that the name is not already being used as a unique name. NetBIOS supports multicast by allowing applications to bind to a group name and communicate.

Topics

Getting Started with Sockets Over NetBIOS
NetBIOS Address Format

Getting Started with Sockets Over NetBIOS

This section provides some basic information for getting started with sockets over NetBIOS:

- Use PF_NETBIOS or PF_NB for the protocol family.
- Use AF_NETBIOS or AF_NB for the address family.
- The following socket types are supported for the NetBIOS domain:
  - Datagram (SOCK_DGRAM)
  - Sequenced packet (SOCK_SEQPACKET)

  The socket type is passed as a parameter to the socket() call. For additional information see Socket Types.
- If a connect() socket call is received without an explicit bind() call, an implicit bind is automatically performed. In this case, the application does not care about its own name, and a unique NetBIOS name is generated by networking services. You can retrieve the NetBIOS name by using the getsockname() call.
- Applications using the NetBIOS communication domain can use sockets in both a connection-oriented (sequenced packet) and connectionless (datagram) mode.
- A NetBIOS application on one workstation can use sockets to communicate with an NCB NetBIOS application on a different workstation.

NetBIOS Address Format

A socket address in a NetBIOS address family is composed of the six fields in the following sockaddr_nb structure: length, address family, address type, a reserved field, adapter number, and NetBIOS name. This structure is located in the <NETNB\NB.H> header file:

```
struct       sockaddr_nb {
    u_char   snb_len;                 /* sizeof(struct sockaddr_nb) */
    u_char   snb_family;              /* AF_NETBIOS */
    short    snb_type;                /* 0:unique or 1:group */
    char     snb_nbnetid[NB_NETIDLEN]; /* NetBIOS netid */
    char     snb_name[NAMELEN];        /* NetBIOS name */
}
```

The length field (*snb_len*) is set to sizeof(struct sockaddr_nb).

The family field (*snb_family*) is set to AF_NETBIOS or AF_NB.

The address type field (*snb_type*) is used to specify the name as either a unique (NB_UNIQUE) or a group (NB_GROUP) name.

The network identifier field (*sub_netid*) is used by the protocol stack to contain the NetBIOS netid, which is the logical adapter number that the name is associated with.

The name field (*snb_name*) contains the 16-byte NetBIOS name, and is used as is.

If a connect() socket call is received without an explicit bind() call, an implicit bind is automatically performed. The application can use any name, and a unique NetBIOS name is generated by the system. A NetBIOS name is generated for this socket by converting the 6-byte MAC address to an ASCII hex string, and postpended with a 2-byte number that increments after each use. You can retrieve the NetBIOS name by using the getsockname() call.

Note that for the NetBIOS domain, more than one socket can be bound to the same local address to establish multiple connections to one or more remote destinations. To enable this feature, the socket option SO_REUSEADDR must be set (see setsockopt()). In addition, you can bind more than one address to the same adapter.

Windows Sockets Version 1.1 for OS/2

This section presents information for implementing OS/2 applications that use the Windows Sockets, or Winsock, API. OS/2 supports two varieties of the Winsock API: one for use by 32-bit Developer API Extensions (Open32) applications, and another for use by 32-bit PM applications and 32-bit Command Line (VIO) applications. Except for the specific differences listed in this section, these implementations comply with the *Windows Sockets Version 1.1* specification, and corrections or amplifications as described in the draft *Windows Sockets Version 2.0* specification.

Topics

Overview
Blocking Hook Support
Asynchronous Functions
Differences between OS/2 WInsock and the Winsock 1.1 Specification
Using the Winsock 1.1 for OS/2 Trace Command and Trace Formatter
Where to Find Winsock Reference Information

Overview

Winsock for OS/2 consists of two DLLs:

PMWSOCK.DLL	for the 32-bit PM and 32-bit VIO environments
DAPWSOCK.DLL	for the Open32 environment

Two header files, <PMWSOCK.H> and <WINSOCK.H>, which are tailored to the PM and Open32 environments respectively, are included. <PMWSOCK.H> is similar to <WINSOCK.H> but has changes to data types and function prototypes to accommodate PM. <PMWSOCK.H> is also used by VIO applications.

Blocking Hook Support

Blocking hooks were introduced in Winsock to support Windows 3.x's non-preemptive multitasking. When an application issues a blocking sockets call, Winsock starts processing the call and then enters a loop where it alternates between calling a blocking hook and checking to see if the socket call has completed or been cancelled. The blocking hook is responsible for dispatching messages to keep the system responsive while the socket call is processing. Blocking hooks are installed on a per-thread basis.

In Win32 systems, blocking hooks aren't necessary because Win32 systems support preemptive multitasking. Consequently, Win32 versions of Winsock do not provide a default blocking hook, and using blocking hooks is discouraged. Nonetheless, an application programmer can call WSASetBlockingHook to install one.

In the OS/2 implementation of Winsock, when a call is made to a Winsock function that blocks and a blocking hook has been installed in that thread, a new thread is started to issue the socket call. The original thread then starts spinning in a loop, alternating between calling the user's blocking-hook function and checking to see if the socket call has completed or has been cancelled. A half-second sleep is inserted between each iteration of the loop to avoid consuming large amounts of CPU time. When the socket call completes, or is cancelled, the new thread passes the results of the socket call to the original thread and then ends.

When a call is made to a Winsock function that blocks and a blocking hook has not been installed for that thread, the socket call is issued directly and no threads are started.

Asynchronous Functions

Asynchronous functions were added to Winsock to make sockets friendlier to the GUI programmer. When an application issues an asynchronous Winsock call, Winsock starts processing the call and immediately returns to the caller. When the call completes, Winsock posts a message to the application's message queue to inform the application of the results of the function.

For the OS/2 implementation of Winsock, asynchronous calls are handled similarly to the way blocking hooks are handled. Each call to an asynchronous Winsock function causes a secondary thread to issue the socket call. The original thread then returns to the application. When the socket call returns, the secondary thread posts a message to the appropriate message queue.

To avoid creating and terminating many threads, the asynchronous calls maintain a small pool of threads that are dedicated to servicing asynchronous calls. The threads are created as needed and are not terminated until the application ends. When the application issues an asynchronous call, Winsock determines if there is a free thread in the pool. If a free thread exists, Winsock uses it to service the call. If there is not a free thread in the pool, Winsock checks to see if the maximum number of threads that can fit in the pool has been started. If the maximum number of threads has not been started, Winsock starts a new thread, adds it to the pool, and has the new thread service the socket call.

If no free threads are in the pool and the pool is filled to capacity, Winsock starts a new thread to service the call. After the call completes and the message has been posted, the new thread terminates.

Differences between OS/2 WInsock and the Winsock 1.1 Specification

This section describes the differences between the TCP/IP for OS/2 implementation of Winsock and the Winsock 1.1 specification.

WSASetLastError and WSAGetLastError APIs

The Winsock specification states that on Win32 systems, WSAGetLastError and WSASetLastError will simply call the Win32 GetLastError and SetLastError functions. For reasons of efficiency, the Open32 version of OS/2 Winsock implements these two calls internally and does not call GetLastError or SetLastError. Also because PM does not have a function that is equivalent to SetLastError, the PM version of Winsock also implements these calls internally.

BSD Compatibility Files

The Winsock 1.1 specification states that the standard <BSD NETDB.H>, <ARPA/INET.H>, <SYS/TIME.H>, <SYS/SOCKET.H>, and <NETINET/IN.H> header files should be supplied by Winsock implementations and should just include <WINSOCK.H>. Because TCP/IP for OS/2 also supplies a BSD socket library, it does not comply with the specification in this regard. If a developer includes any of those header files, the developer will get the BSD version of those header files, not the Winsock version.

Calling Conventions

In the Winsock 1.1 specification, Winsock calls are defined to use the PASCAL calling convention (the standard calling convention on Windows systems). OS/2 also has the PASCAL calling convention, but instead it uses APIENTRY as the standard calling convention. Because of this, OS/2 Winsock uses the APIENTRY convention for all of its Winsock calls.

In the Open32 environment, PASCAL is defined (with #define) to be APIENTRY by <OS2WDEF.H>, which is part of Open32, so no changes are needed to <WINSOCK.H> to accommodate APIENTRY.

In the PM and VIO environments, PASCAL and APIENTRY are two different calling conventions, so the Winsock function prototypes in <PMWSOCK.H> have been modified by replacing all occurrences of PASCAL with APIENTRY.

Topics

Differences between <PMWSOCK.H> and Standard <WINSOCK.H> Header Files
Differences between IBM Open32 <WINSOCK.H> and Standard <WINSOCK.H> Header Files

----

Differences between and Standard Header Files

Following are the differences between <PMWSOCK.H> and standard <WINSOCK.H> header files:

- <PMWSOCK.H> does not include (with #include) any header files.
- The keyword FAR has been removed from all functions and pointers.
- The keyword PASCAL has been removed from all of the prototypes and replaced with APIENTRY.
- The FARPROC type in the WSASetBlockingHook prototype has been replaced with PFN.
- The hostent, netent, servent, and protoent structures had several entries that were defined as "short;" these have been widened to "int." The two fields of the linger structure have been widened from "u_short" to "int." These changes were made to comply with the definitions in the BSD header files.

- A bug in the IN_CLASSC macro has been corrected.
- The CONST keyword for the *buf* parameter of the prototype for WSAAsyncGetHostByAddr has been removed (it was an error).

Differences between IBM Open32 and Standard Header Files

Following are the differences between IBM Open32 and <WINSOCK.H>:

- <WINSOCK.H> includes <OS2WIN.H> instead of <WINDOWS.H>.
- The keyword FAR has been removed from all functions and pointers.
- The hostent, netent, servent, and protoent structures had several entries that were defined as "short." These have been widened to "int." The two fields of the linger structure have been widened from "u_short" to "int." These changes were made to comply with the definitions in the BSD header files.
- A bug in the IN_CLASSC macro has been corrected.
- The CONST keyword for the *buf* parameter of the prototype for WSAAsyncGetHostByAddr has been removed (it was an error).

Using the Winsock 1.1 for OS/2 Trace Command and Trace Formatter

Winsock 1.1 for OS/2 includes tracing capability to aid Winsock developers in isolating problems related to one of the following:

- The client application
- The Winsock 1.1 DLL

Topics

Sample Winsock Trace Output
WSTRACE Command
WSFORMAT Command

Sample Winsock Trace Output

Using the trace capability, developers can trace Winsock procedure calls and exits, parameter values, and return values. Following is a sample of formatted Winsock trace output:

```
Winsock Trace - Version 1   Trace Date 05/07/1996   Trace Time 14:04:51.28
Thread  TimeStamp  Winsock Function  (Parameters)
-------------------------------------------------------------------------------

00001 14:04:51.35 WSAISBLOCKING Entry  ()
00001 14:04:51.35 WSAISBLOCKING Exit  (False : No blocking call in progress)
00001 14:04:51.35 WSASTARTUP Exit  (wVersionRequired=101, lpWSADATA=4B00C Return=0)
00001 14:04:51.35 GETHOSTNAME Entry ()
00001 14:04:51.35 WSAISBLOCKING Entry  ()
00001 14:04:51.35 WSAISBLOCKING Exit  (False : No blocking call in progress)
```

```
00001 14:04:51.35 GETHOSTNAME Exit (Return=0, Hostname=screamin)
00001 14:04:51.38 INET_NTOA Entry  (in_addr=4263781641)
00001 14:04:51.38 INET_NTOA Exit  (in_addr=4263781641, Return=9.37.36.254)
00001 14:04:57.25 SOCKET Entry  (address family=2[af_inet], type=1[sock_stream], protocol=0[ip])
00001 14:04:57.25 WSAISBLOCKING Entry  ()
00001 14:04:57.25 WSAISBLOCKING Exit  (False : No blocking call in progress)
00001 14:04:57.25 SOCKET Exit  (address family=2[af_inet], type=1[sock_stream], protocol=0[ip] Return=187)
00001 14:05:05.88 GETHOSTNAME Entry ()
00001 14:05:05.88 WSAISBLOCKING Entry  ()
00001 14:05:05.88 WSAISBLOCKING Exit  (False : No blocking call in progress)
00001 14:05:05.88 GETHOSTNAME Exit (Return=0, Hostname=screamin)
00001 14:05:11.03 HTONS Entry  (host=0)
00001 14:05:11.03 HTONS Exit  (host=0, Return=0)
00001 14:05:11.03 BIND Entry  (socket=187, sockaddr struct[family=2 (af_inet), port=0, s_addr=9.37.36.254])
00001 14:05:11.03 WSAISBLOCKING Entry  ()
00001 14:05:11.03 WSAISBLOCKING Exit  (False : No blocking call in progress)
00001 14:05:11.03 BIND Exit  (socket=187, sockaddr struct[family=2 (af_inet), port=0, s_addr=9.37.36.254] Retur
00001 14:05:19.31 CLOSESOCKET Entry  (Socket=187)
00001 14:05:19.31 WSAISBLOCKING Entry  ()
00001 14:05:19.31 WSAISBLOCKING Exit  (False : No blocking call in progress)
00001 14:05:19.31 CLOSESOCKET Exit  (Socket=187, Return=0)
00001 14:05:28.91 WSACLEANUP Entry ()
00001 14:05:28.91 WSAISBLOCKING Entry  ()
00001 14:05:28.91 WSAISBLOCKING Exit  (False : No blocking call in progress)
```

The following sections provide details for using the Winsock 1.1 for OS/2 trace command and trace formatter.

WSTRACE Command

The WSTRACE command enables and disables Winsock tracing in an OS/2 session.

Syntax

```
                on
   WSTRACE
                off
                            filename
                            COM1
                            COM2
                            COM3
                            COM4
                                 WSTRACE
                           -p
                                 pipename

        64
                                                                        `
         -b buffsize
```

Parameters

on
: Turns tracing on. This is the default.

off
: Turns tracing off.

filename
: Specifies the file name of the file that the trace output will be written to. COM1, COM2, COM3, and COM4 may be used to direct the trace to a serial port.

-p *pipename*
: Specifies the pipe name of an OS/2 pipe that the trace output will be written to. Pipe names do not have to include a \PIPE\ prefix. If *pipename* is not specified with the -p parameter, the pipe name defaults to WSTRACE.

-b *buffsize*

Specifies the size of the global trace buffer. If the -b parameter is not specified, the buffer size defaults to 64 KB.

Notes:

1. When neither *filename* nor *pipename* is specified, trace output is written by default to the file WSTRACE.DMP in the local directory.

2. When a pipe or COM port is specified as the destination for tracing information, start the WSFORMAT command in an OS/2 session on the destination device *before* running the Winsock application to be traced.

3. When specifying a serial port as the Winsock trace device, be sure that the serial port settings match on both ends of the line. The OS/2 MODE command lets you set the serial port settings. For more information about the MODE command, see the *OS/2 Command Reference* or enter:

   ```
   help mode
   ```

   in an OS/2 session.

WSFORMAT Command

The WSFORMAT command is the Winsock trace formatter. The WSFORMAT command accepts the binary trace input from a serial port, file, or OS/2 pipe, and converts the input into a readable format that can then be displayed on-screen or written to a file, serial port, or OS/2 pipe.

Syntax

```
WSFORMAT

                input_filename
                COM1
                COM2
                COM3
                COM4
                     WSTRACE
               -p
                     input_pipename

     WSFORMAT.DMP

      -f formatted_output_filename

     WSTRACE.DMP

      -b binary_output_filename
```

Parameters

input_filename
Specifies the file name of the input file containing the trace information to format. COM1, COM2, COM3, and COM4 may be used to accept trace data from a serial port.

-p *input_pipename*
Specifies the pipe name of the OS/2 pipe that contains the trace data to format. Pipe names do not have to include a \PIPE\ prefix. If *input_pipename* is not specified with the -p parameter, the pipe name defaults to WSTRACE.

-f *formatted_output_file*
Specifies the file name of the file that the formatted trace output will be written to. If *formatted_output_file* is not specified with the -f parameter, the file name defaults to WSFORMAT.DMP in the current directory.

-b *binary_output_file*
Specifies the file name of the file that will receive a copy of the binary information being formatted. This can be used to save a copy of the trace information coming from a serial port or pipe. If *binary_output_file* is not specified with the -b parameter, the output file name defaults to WSTRACE.DMP in the local directory.

Notes:

1. If the -f option is not specified, the formatted trace output is written to the screen.

2. When specifying a serial port as the Winsock trace device, be sure that the serial port settings match on both ends of the line. The OS/2 MODE command lets you set the serial port settings. For more information about the MODE command, see the *OS/2 Command Reference* or enter:

   ```
   help mode
   ```

   in an OS/2 session.

Where to Find Winsock Reference Information

To obtain Winsock reference information, including socket and database routines and Microsoft Windows-specific extensions, consult the *Windows Sockets* specification. This specification is documented by authors from a number of corporations and is available from many places, including:

- The IBM TCP/IP for OS/2 Programmer's Development Toolkit on the IBM Developer's Connection CD-ROM (*Windows Sockets* Version 1.1 only)
- The Internet, at URL *http://www.stardust.com* (*Windows Sockets* Version 1.1 only)
- Directory `/pub/winsock` on host `microdyne.com` by way of anonymous ftp

To be added to the *Windows Sockets* mailing list, address your request to *winsock-request@microdyne.com.*

Remote Procedure Calls

This section describes the high-level remote procedure calls (RPCs) implemented in TCP/IP for OS/2, including the RPC programming interface to the C language and communication between processes.

Topics

The RPC Protocol
The RPC Interface
Remote Programs and Procedures
Portmapper
eXternal Data Representation (XDR)
The RPC Intermediate Layer
The RPC Lowest Layer
rpcgen Command
rpcinfo Command
The enum clnt_stat Structure
The Remote Procedure Call Library
Differences between OS/2 and Sun Microsystems RPCs
Compiling an RPC API Application

The RPC Protocol

The RPC protocol enables remote execution of subroutines across a TCP/IP network. RPC, together with the eXternal Data Representation (XDR) protocol, defines a standard for representing data that is independent of internal protocols or formatting. RPCs can communicate between processes on the same or different hosts.

The RPC protocol enables users to work with remote procedures as if the procedures were local. The remote procedure calls are defined through routines contained in the RPC protocol. Each call message is matched with a reply message. The RPC protocol is a message-passing protocol that implements other non-RPC protocols, such as batching and broadcasting remote calls. The RPC protocol also supports callback procedures and the select subroutine on the server side.

RPC provides an authentication process that identifies the server and client to each other. RPC includes a slot for the authentication parameters on every remote procedure call so that the caller can identify itself to the server. The client package generates and returns authentication parameters. RPC supports various types of authentication, such as the UNIX systems.

In RPC, each server supplies a program that is a set of procedures. The combination of a host address, a program number, and a procedure number specifies one remote service procedure. In the RPC model, the client makes a procedure call to send a data packet to the server. When the packet arrives, the server calls a dispatch routine, performs whatever service is requested, and sends a reply back to the client. The procedure call then returns to the client.

RPC is divided into two layers: intermediate and lowest. Generally, you use the RPC interface to communicate between processes on different workstations in a network. However, RPC works just as well for communication between different processes on the same workstation.

The Portmapper program maps RPC program and version numbers to a transport-specific port number. The Portmapper program makes dynamic binding of remote programs possible.

To write network applications using RPC, programmers need a working knowledge of network theory and C programming language. For most applications, understanding the RPC mechanisms usually hidden by the RPCGEN protocol compiler is also helpful. However, RPCGEN makes understanding the details of RPC unnecessary. The figures in The RPC Interface give an overview of the high-level RPC client and server processes from initialization through cleanup. See the SAMPLES\RPC directory for sample RPC client, server, and raw data stream programs. RPCGEN samples are in the SAMPLES\RPCGEN directory.

For more information about the RPC and XDR protocols, see the Sun Microsystems publication, *Networking on the Sun Workstation: Remote Procedure Call Programming Guide* , RFC 1057 and RFC 1014.

The RPC Interface

The RPC model is similar to the local procedure call model. In the local model, the caller places the argument to a procedure in a specified location such as a result register. Then, the caller transfers control to the procedure. The caller eventually regains control, extracts the results of the procedure, and continues the execution.

RPC works in the same way: One thread of control winds logically through the caller and server processes as follows:

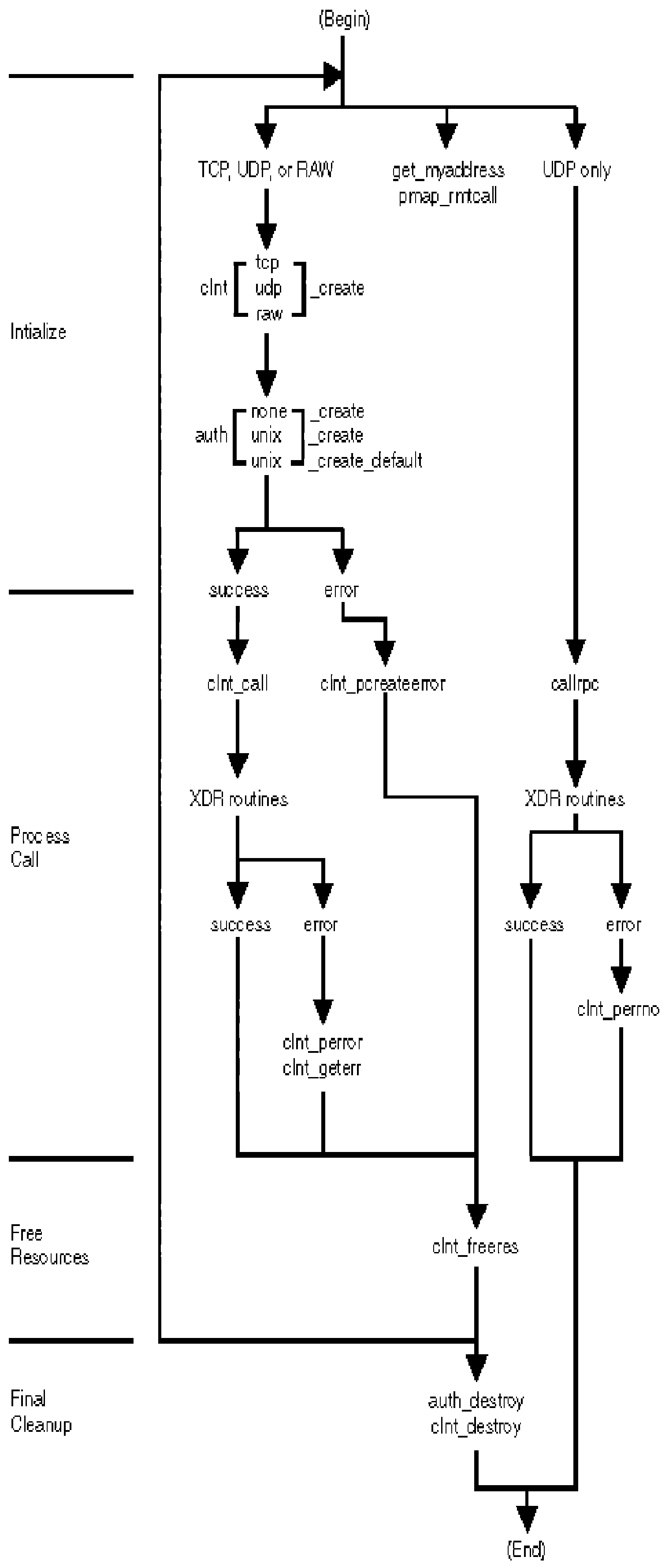
1. The caller process sends a call message that includes the procedure parameters to the server process and then waits for a reply message (blocks).

2. A process on the server side, which is dormant until the arrival of the call message, extracts the procedure parameters, computes the results, and sends a reply message. Then the server waits for the next call message.

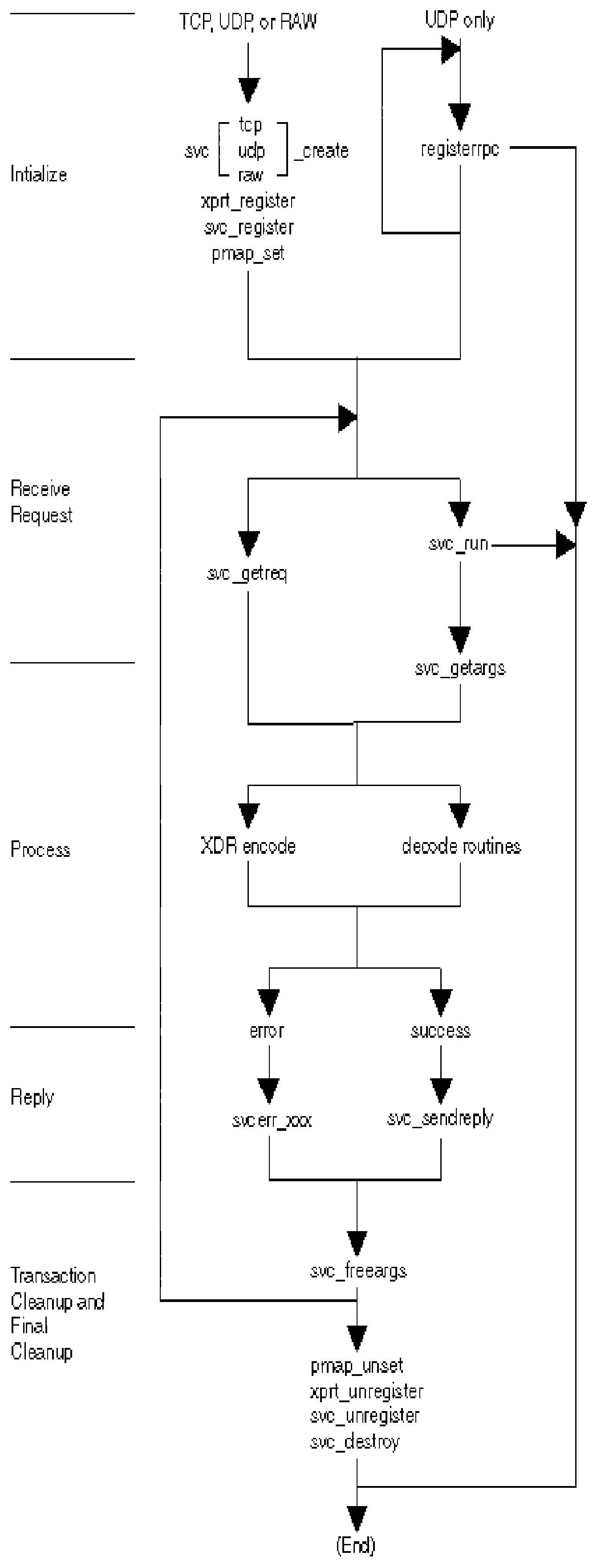
3. A process on the caller side receives the reply message and extracts the results of the procedure. The caller then resumes the execution.

See the following figures for an illustration of the RPC model:

Remote Procedure Call (Client)


(Begin)
TCP, UDP, or RAW
get_myaddress
pmap_rmtcall
UDP only
clnt
tcp
udp
raw
_create
Intialize
auth
none
unix
unix
_create
_create
_create_default
success
error
clnt_call
clnt_pcreateerror
callrpc
XDR routines
XDR routines
Process
Call
success
error
success
error
clnt_perrno
clnt_perror
clnt_geterr
Free
Resources
clnt_freeres
Final
Cleanup
auth_destroy
clnt_destroy
(End)

Remote Procedure Call (Server)



Remote Programs and Procedures

The RPC call message has three unsigned fields:

- Remote program number
- Remote program version number
- Remote procedure number

The three fields uniquely identify the procedure to be called. The program number defines a group of related remote procedures, each of which has a different procedure number. Each program also has a version number.

The central system authority administers the program number. A remote program number is assigned by groups of 0x20000000, as shown in the following list:

Program Number	**Description**
0-1xxxxxxx	Is predefined and administered by the OS/2 TCP/IP system.
20000000-3xxxxxxx	Represents the user defined numbers
40000000-5xxxxxxx	Represents transient numbers
60000000-7xxxxxxx	Reserved
80000000-9xxxxxxx	Reserved
a0000000-bxxxxxxx	Reserved
c0000000-dxxxxxxx	Reserved
e0000000-fxxxxxxx	Reserved

Portmapper

This section describes the Portmapper service and its uses.

Topics

Portmapper Protocol
Registering and Unregistering a Port with Portmapper
Contacting Portmapper
Portmapper Procedures

Portmapper Protocol

The Portmapper protocol defines a network service that clients use to look up the port number of any remote program supported by the server. The client programs must find the port numbers of the server programs that they intend to use.

The Portmapper program:

- Maps RPC program and version numbers to transport specific port numbers.

- Enables dynamic binding of remote programs. This is desirable because the range of reserved port numbers is small, and the number of potential remote programs is large. When running only the Portmapper program on a reserved port, you can determine the port numbers of other remote programs by querying Portmapper.

- Supports both the UDP and TCP protocols.

The RPC client contacts Portmapper on port number 111 on either of these protocols.

Registering and Unregistering a Port with Portmapper

Portmapper is the only network service that must have a dedicated port (111). Other RPC network services can be assigned port numbers statically or dynamically, if the services register their ports with the host's local Portmapper. The RPC server can register or unregister their services by using the following calls:

svc_register()	Associates a program with the service dispatch routine
svc_unregister()	Removes all local mappings to dispatch routines and port numbers
registerrpc()	Registers a procedure with the local Portmapper and creates a control structure to remember the server procedure and its XDR routine

----

Contacting Portmapper

To find the port of a remote program, the client sends an RPC request to well-known port 111 of the server's host. If Portmapper has a port number entry for the remote program, Portmapper provides the port number in the RPC reply. The client then requests the remote program by sending an RPC request to the port number provided by Portmapper.

Clients can save port numbers of recently called remote programs to avoid having to contact Portmapper for each request to a server.

RPC also provides the following calls for interfacing with Portmapper:

Call	**Description**
pmap_getmaps()	Returns a list of current program-to-port mappings on the foreign host
pmap_getport()	Returns the port number associated with the remove program, version, and transport protocol
pmap_rmtcall()	Instructs Portmapper to make an RPC call to a procedure on the host
pmap_set()	Sets the mapping of a program to a port on the local machine
pmap_unset()	Removes mappings associated with the program and version number on the local machine
xdr_pmap()	Translates an RPC procedure identification
xdr_pmaplist()	Translates a variable number of RPC procedure identifications

----

Portmapper Procedures

The Portmapper program supports the following procedures:

Procedure	**Description**
PMAPPROC_NULL	Has no parameters. A caller can use the return code to determine if Portmapper is running.
PMAPPROC_SET	Registers itself with the Portmapper program on the same machine. It passes the:

	• Program number • Program version number • Transport protocol number • Port number The procedure has successfully established the mapping if the return value is TRUE. The procedure does not establish a mapping if one already exists.
PMAPPROC_UNSET	Unregisters the program and version numbers with Portmapper on the same machine.
PMAPPROC_GETPORT	Returns the port number when given a program number, version number, and transport protocol number. A port value of 0 indicates the program has not been registered.
PMAPPROC_DUMP	Takes no input, but returns a list of program, version, protocol, and port numbers.
PMAPPROC_CALLIT	Allows a caller to call another remote procedure on the same machine without knowing the remote procedure's port number. The PMAPPROC_CALLIT procedure sends a response only if the procedure is successfully run.

----

eXternal Data Representation (XDR)

This section describes the eXternal Data Representation (XDR) standard and its use.

Topics

The XDR Standard
Basic Block Size
The XDR Subroutine Format
XDR Data Types and their Filter Primitives
XDR Filter Primitives
XDR Nonfilter Primitives

The XDR Standard

An eXternal Data Representation (XDR) is a data representation standard that is independent of languages, operating systems, manufacturers, and hardware architecture. This standard enables networked computers to share data regardless of the machine on which the data is produced or consumed. The XDR language permits transfer of data between diverse computer architectures.

An XDR approach to standardizing data representations is canonical. That is, XDR defines a single byte (big endian), a single floating-point representation (IEEE), and so on. Any program running on any machine can use XDR to create portable data by translating its local representation to the XDR standards. Similarly, any program running on any machine can read portable data by translating the XDR standard representations to its local equivalents.

The XDR standard is the backbone of the RPC, because data for remote procedure calls is sent using the XDR standard.

To use XDR routines, C programs must include the <RPC\XDR.H> header file, which is automatically included by the <RPC\RPC.H> header file.

Basic Block Size

The XDR language is based on the assumption that bytes (an octet) can be ported to, and encoded on, media that preserve the meaning of the bytes across the hardware boundaries of data. XDR does not represent bit fields or bit maps; it represents data in blocks of multiples of 4 bytes (32 bits). If the bytes needed to contain the data are not a multiple of four, enough (1 to 3) bytes to make the total byte count a multiple of four follow the *n* bytes. The bytes are read from, or written to, a byte stream in order. The order dictates that byte *m* precedes *m*+1. Bytes are ported and encoded from low order to high order in local area networks (LANs). Representing data in standardized formats resolves situations that occur when different byte-ordering formats exist on networked machines. This also enables machines with different structure-alignment algorithms to communicate with each other.

The XDR Subroutine Format

An XDR routine is associated with each data type. XDR routines have the following format:

```
xdr_xxx(xdrs,dp)
          XDR *xdrs;
          xxx *dp;
{
}
```

The routine has the following parameters:

xxx
XDR data type.

xdrs
Opaque handle that points to an XDR stream. The system passes the opaque handle pointer to the primitive XDR routines.

dp
Address of the data value that is to be encoded or decoded.

If they succeed, the XDR routines return a value of 1; if they do not succeed, they return a value of 0.

XDR Data Types and their Filter Primitives

The XDR standard defines basic and constructed data types. The XDR filter primitives are routines that define the basic and constructed data types. The XDR language provides RPC programmers with a specification for uniform representation that includes filter primitives for basic and constructed data types.

The basic data types include:

- Integers
- Enumeration
- Booleans
- Floating point decimals
- Void
- Constants
- Typedef
- Optional data

The constructed data types include:

- Arrays
- Opaque data
- Strings
- Byte arrays
- Structures
- Discriminated unions
- Pointers

XDR Filter Primitives

The XDR standard translates both basic and constructed data types. For basic data types such as integer, XDR provides basic filter primitives that:

- Serialize information from the local host's representation to XDR representation
- Deserialize information from the XDR representation to the local host's representation

For constructed data types, XDR provides constructed filter primitives that allow the use of basic data types (such as integers and floating-point numbers) to create more complex constructs (such as arrays and discriminated unions).

Topics

Integer Filter Primitives
Enumeration Filter Primitives
Floating-Point Filter Primitives
Opaque Data Filter Primitive
Array Filter Primitives
String Filter Primitives
Primitive for Pointers to Structures
Primitive for Discriminated Unions
Passing Routines without Data

Integer Filter Primitives

The XDR filters cover signed and unsigned integers, as well as signed and unsigned short and long integers.

The routines for XDR integer filters are:

Routine	Description
xdr_int()	Translates between C integers and their external representations
xdr_u_int()	Translates between C unsigned integers and their external representations
xdr_long()	Translates between C long integers and their external representations
xdr_u_long()	Translates between C unsigned long integers and their external representations
xdr_short()	Translates between C short integers and their external representations
xdr_u_short()	Translates between C unsigned short integers and their external representations

----

Enumeration Filter Primitives

The XDR library provides a primitive for generic enumerations based on the assumption that a C enumeration value (enum) has the same representation. A special enumeration in XDR, known as the *Boolean* , provides a value of 0 or 1 represented internally in a binary notation.

The routines for the XDR library enumeration filters are:

Routine	Description
xdr_enum()	Translates between C language enums and their external representations
xdr_bool()	Translates between Booleans and their external representations

----

Floating-Point Filter Primitives

The XDR library provides primitives that translate between floating-point data and their external representations. Floating-point data encodes an integer with an exponent. Floats and double-precision numbers compose floating-point data.

Note: Numbers are represented as Institute of Electrical and Electronics Engineers (IEEE) standard floating points. Routines might fail when decoding IEEE representations into machine specific representations.

The routines for the XDR floating-point filters are:

Routine	**Description**
xdr_float()	Translates between C language floats and their external representations
xdr_double()	Translates between C language double-precision numbers and their external representations

----

Opaque Data Filter Primitive

Opaque data is composed of bytes of a fixed size that are not interpreted as they pass through the data streams. Opaque data bytes, such as handles, are passed between server and client without being inspected by the client. The client uses the data as it is and then returns it to the server. By definition, the actual data contained in the opaque object is not portable between computers.

The XDR library includes the following routine for opaque data:

Routine	**Description**
xdr_opaque()	Translates between opaque data and its external representation

----

Array Filter Primitives

Arrays are constructed filter primitives that can be generic arrays or byte arrays. The XDR library provides filter primitives for handling both types of arrays.

Generic Arrays

These consist of arbitrary elements. You use them in much the same way as byte arrays. The primitive for generic arrays requires an additional parameter to define the size of the element in the array and to call an XDR routine to encode or decode each element in the array.

The XDR library includes the following routines for generic arrays:

Routine	**Description**
xdr_array()	Translates between variable-length arrays and their corresponding external representations
xdr_vector()	Translates between fixed-length arrays and their corresponding external representations

----

Byte Arrays

These differ from strings by having a byte count. That is, the length of the array is set to an unsigned integer. They also differ in that byte arrays do not end with a null character. The XDR library provides a primitive for byte arrays. External and internal representations of byte arrays are the same.

The XDR library includes the following routine for byte arrays:

Routine	**Description**
xdr_bytes()	Translates between counted byte string arrays and their external representations

----

String Filter Primitives

A string is a constructed filter primitive that consists of a sequence of bytes terminated by a null byte. The null byte does not figure into the length of the string. Externally, strings are represented by a sequence of American Standard Code Information Interchange (ASCII) characters. Internally, XDR represents them as pointers to characters with the designation `char *`.

The XDR library includes primitives for the following string routines:

Routine	**Description**
xdr_string()	Translates between C language strings and their external representations
xdr_wrapstring()	Calls the xdr_string subroutine

Primitive for Pointers to Structures

The XDR library provides the primitive for pointers so that structures referenced within other structures can be easily serialized, deserialized, and released.

The XDR library includes the following routine for pointers to structures:

Routine	**Description**
xdr_reference()	Provides pointer chasing within structures

Primitive for Discriminated Unions

A discriminated union is a C language union, which is an object that holds several data types. One arm of the union contains an enumeration value (enum_t), or discriminant, that holds a specific object to be processed over the system first.

The XDR library includes the following routine for discriminated unions:

Routine	**Description**
xdr_union()	Translates between discriminated unions and their external representations

----

Passing Routines without Data

Sometimes an XDR routine must be supplied to the RPC system, but no data is required or passed. The XDR library provides the following primitive for this function:

Routine	Description
xdr_void()	Supplies an XDR subroutine to the RPC system without sending data

----

XDR Nonfilter Primitives

Use the XDR nonfilter primitives to create, manipulate, implement, and destroy XDR data streams. These primitives allow you to:

- Describe the data stream position
- Change the data stream position
- Destroy a data stream

Topics

Creating and Using XDR Data Streams
Manipulating an XDR Data Stream
Implementing an XDR Data Stream
Destroying an XDR Data Stream

Creating and Using XDR Data Streams

You get XDR data streams by calling creation routines that take arguments specifically designed for the properties of the stream. There are existing XDR data streams for serializing or deserializing data in standard input and output streams, memory streams, and record streams.

Note: RPC clients do not have to create XDR streams, because the RPC system creates and passes these streams to the client.

The types of data streams include:

- Standard I/O streams
- Memory streams
- Record streams

Standard I/O Streams

XDR data streams serialize and deserialize standard input/output(I/O) by calling the standard I/O creation routine to initialize the XDR data stream pointed to by the *xdrs* parameter.

The XDR library includes the following routine for standard I/O data streams:

Routine	Description
xdrstdio_create()	Initializes the XDR data stream pointed to by the *xdrs* parameter

----

Memory Streams

XDR data streams serialize and deserialize data from memory by calling the XDR memory creation routine to initialize, in local memory, the XDR stream pointed at by the *xdrs* parameter. In RPC, the UDP/IP implementation of remote procedure calls uses this routine to build entire call and reply messages in memory before sending the message to the recipient.

The XDR library includes the following routine for memory data streams:

Routine	Description
xdrmem_create()	Initializes, in local memory, the XDR stream pointed to by the *xdrs* parameter

----

Record Streams

Record streams are XDR streams built on top of record fragments, which are built on TCP/IP streams. TCP/IP is a connection protocol for transporting large streams of data at one time rather than transporting a single data packet at a time.

The primary use of a record stream is to interface remote procedure calls to TCP connections. It can also be used to stream data into or out of normal files.

XDR provides the following routines for use with record streams:

Routine	Description
xdrrec_create()	Provides an XDR stream that can contain long sequences of records
xdrrec_endofrecord()	Causes the current outgoing data to be marked as a record
xdrrec_skiprecord()	Causes the position of an input stream to move to the beginning of the next record
xdrrec_eof()	Checks the buffer for an input stream that identifies the end of file (EOF)

Manipulating an XDR Data Stream

XDR provides the following routines for describing the data stream position and changing the data stream position:

Routine	Description
xdr_getpos()	Returns an unsigned integer that describes the current position in the data stream
xdr_setpos()	Changes the current position in the XDR stream

Implementing an XDR Data Stream

You can create and implement XDR data streams. The following example shows the abstract data types (XDR handle) required for you to implement your own XDR streams. They contain operations applied to the stream (an operation vector for the particular implementation) and two private fields for using that implementation.

```
enum xdr_op  { XDR_ENCODE=0, XDR_DECODE=1, XDR_FREE=2 };
typedef struct xdr {
        enum xdr_op  x_op;
        struct xdr_ops {
               bool_t  (*x_getlong)(struct xdr *, long *);
               bool_t  (*x_putlong)(struct xdr *, long *);
               bool_t  (*x_getbytes)(struct xdr *, caddr_t, u_int);
        /* get some bytes from " */
               bool_t  (*x_putbytes)(struct xdr *, caddr_t, u_int);
        /* put some bytes to " */
               u_int   (*x_getpostn)(struct xdr *);
               bool_t  (*x_setpostn)(struct xdr *,u_int);
               long *  (*x_inline)(struct xdr *,u_int);
```

```
              void    (*x_destroy)(struct xdr *);
       } *x_ops;
       caddr_t         x_public;
       caddr_t         x_private;
       caddr_t         x_base;
       int             x_handy;
} XDR;
```

The following parameters are pointers to XDR stream manipulation routines:

Parameter	**Description**
x_getlong	Gets long integer values from the data stream.
x_putlong	Puts long integer values into the data stream.
x_getbytes	Gets bytes from the data streams.
x_putbytes	Puts bytes into the data streams.
x_getpostn	Returns the stream offset.
x_setpostn	Repositions the offset.
x_inline	Points to an internal data buffer, used for any purpose.
x_destroy	Frees the private data structure.
x_ops	Specifies the current operation being performed on the stream. This field is important to the XDR primitives, but the stream's implementation does not depend on the value of this parameter.

The following fields are specific to a stream's implementation:

Field	**Description**
x_public	Specific user data that is private to the stream's implementation and that is not used by the XDR primitive
x_private	Points to the private data
x_base	Contains the position information in the data stream that is private to the user implementation
x_handy	Data can contain extra information as necessary

Destroying an XDR Data Stream

XDR provides a routine that destroys the XDR stream pointed to by the *xdrs* parameter and frees the private data structures allocated to the stream.

Routine	**Description**
xdr_destroy()	Destroys the XDR stream pointed to by the *xdrs* parameter

The use of the XDR stream handle is undefined after it is destroyed.

The RPC Intermediate Layer

The calls of the RPC intermediate layer are:

Routine	**Description**
registerrpc()	Registers a procedure with the local Portmapper
callrpc()	Calls a remote procedure on the specified system
svc_run()	Accepts RPC requests and calls the appropriate service using svc_getreq()

The transport mechanism is the User Datagram Protocol (UDP). The UDP transport mechanism handles only arguments and results that are less than 8K bytes in length. At this level, RPC does not allow timeout specifications, choice of transport, or process control, in case of errors. If you need this kind of control, consider the lowest layer of RPC.

With only these three RPC calls, you can write a powerful RPC-based network application. The sequence of events follows:

1. Use the registerrpc() call to register your remote program with the local Portmapper. See Portmapper for more information. The following is an example of an RPC server:

```
/* define remote program number and version */

#define RMTPROGNUM (u_long)0x3fffffffL
#define RMTPROGVER (u_long)0x1
#define RMTPROCNUM (u_long)0x1

#include <stdio.h>
#include <rpc\rpc.h>




main()
{
 int *rmtprog();

 /* register remote program with Portmapper */
 registerrpc(RMTPROGNUM, RMTPROGVER, RMTPROCNUM, rmtprog,
 xdr_int, xdr_int);
 /* infinite loop, waits for RPC request from client */
 svc_run();
 printf("Error: svc_run should never reach this point \n");
 exit(1);
}

int *
rmtprog(inproc)          /* remote program */
int *inproc;

{
int *outproc;
...
 /* Process request */
...
 return (outproc);
}
```

The registerrpc() call registers a C procedure rmtprog, which corresponds to a given RPC procedure number.

The registerrpc() call has six parameters:

- The first three parameters are the program, version, and procedure numbers of the remote procedure to be registered.
- The fourth parameter, rmtprog, is the name of the local procedure that implements the remote procedure.
- The last two parameters, xdr_int, are the XDR filters for the remote procedure's arguments and results.

After registering a procedure, the RPC server goes into an infinite loop waiting for a client request to service.

2. The RPC client uses callrpc() to make a service request to the RPC server. The following is an example of an RPC client using the callrpc() call:

```
/* define remote program number and version */

#define RMTPROGNUM (u_long)0x3fffffffL
#define RMTPROGVER (u_long)0x1
#define RMTPROCNUM (u_long)0x1

#include <stdio.h>
#include <rpc\rpc.h>




main()
{
  int inproc=100, outproc, rstat;

...

  /* service request to host RPCSERVER_HOST */
  if (rstat = callrpc("RPCSERVER_HOST", RMTPROGNUM,
              RMTPROGVER, RMTPROCNUM, xdr_int, (char *)&inproc,
              xdr_int, (char *)&outproc)!= 0)
     {
      clnt_perrno(rstat);   /* Why  callrpc() failed ? */
      exit(1);
     }
...

  }
```

The callrpc() call has eight parameters:

- The first is the name of the remote server machine.
- The next three parameters are the program, version, and procedure numbers.
- The fifth and sixth parameters are an XDR filter, and an argument to be encoded and passed to the remote procedure.
- The final two parameters are a filter for decoding the results returned by the remote procedure, and a pointer to the place where the procedure's results are to be stored.

You handle multiple arguments and results by embedding them in structures. The callrpc() call returns 0 if it succeeds, otherwise nonzero. The exact meaning of the returned code is in the <RPC\CLNT.H> header file and is an enum clnt_stat structure cast into an integer.

The RPC Lowest Layer

This section describes the lowest layer of RPC and when to use it.

Topics

When to Use the RPC Lowest Layer
Server Side Program
Client Side Program

When to Use the RPC Lowest Layer

Use the lowest layer of RPC in the following situations:

- You need to use TCP. The intermediate layer uses UDP, which restricts RPC calls to 8K bytes of data. TCP permits calls to send long streams of data.
- You want to allocate and free memory while serializing or deserializing messages with XDR routines. No RPC call at the intermediate level explicitly permits freeing memory. XDR routines are used for memory allocation as well as for serializing and deserializing.
- You need to perform authentication on the client side or the server side by supplying credentials or verifying them.

Server Side Program

The following is an example of the lowest layer of RPC on the server side program:

```
#define RMTPROGNUM   (u_long)0x3fffffffL
#define RMTPROGVER   (u_long)0x1L
#define LONGPROC   1
#define STRINGPROC 2

#define MAXLEN 100

#include <stdio.h>
#include <rpc\rpc.h>
#include <sys\socket.h>

main(argc, argv)
int argc;
char *argv[ ];

{
     int rmtprog();
     SVCXPRT *transp;

...

/* create TCP transport handle */
     transp = svctcp_create(RPC_ANYSOCK, 1024*10, 1024*10);
/* or create UDP transport handle */
/*   transp = svcudp_create(RPC_ANYSOCK);  */
     if (transp == NULL)   /* check transport handle creation */
      {
        fprintf(stderr, "can't create an RPC server transport\n");
        exit(-1);
      }

/* If exists, remove the mapping of remote program and port */
     pmap_unset(RMTPROGNUM, RMTPROGVER);

/* register remote program (TCP transport) with local Portmapper */
     if (!svc_register(transp, RMTPROGNUM, RMTPROGVER, rmtprog,
                         IPPROTO_TCP))
/* or register remote program (UDP transport) with local Portmapper */
/*   if (!svc_register(transp, RMTPROGNUM, RMTPROGVER, rmtprog,*/
                      /* IPPROTO_UDP)) */
      {
       fprintf(stderr, "can't register rmtprog() service\n");
       exit(-1);
      }

     svc_run();
     printf("Error:svc_run should never reaches this point \n");
     exit(1);

}

rmtprog(rqstp, transp)           /* code for remote program */
struct svc_req *rqstp;
SVCXPRT *transp;
{
   long in_long,out_long;
   char buf[100], *in_string=buf, *out_string=buf;
...
   switch((int)rqstp->rq_proc)   /* Which procedure ? */
   {
    case NULLPROC:
         if (!svc_sendreply(transp,xdr_void, 0))
          {
           fprintf(stderr,"can't reply to RPC call\n");
           exit(-1);
          }
         return;

    case LONGPROC:
...
         /* Process the request */
         if (!svc_sendreply(transp,xdr_long,&out_long))
          {
           fprintf(stderr,"can't reply to RPC call\n");
           exit(-1);
          }
         return;

    case STRINGPROC:   /* send received "Hello" message back */
```

```
                                /* to client */
           svc_getargs(transp,xdr_wrapstring,(char *)&in_string);
           strcpy(out_string,in_string);

           /* send a reply back to a RPC client */
           if (!svc_sendreply(transp,xdr_wrapstring,
                                             (char *)&out_string))
            {
             fprintf(stderr,"can't reply to RPC call\n");
             exit(-1);
            }
           return;
     case ... :
...
         /* Any Remote procedure in RMTPROGNUM program */
...
     default:
         /* Requested procedure not found */
         svcerr_noproc(transp);
         return;
     }
}
```

The following steps describe the lowest layer of RPC on the server side program:

1. Service the transport handle.

   The svctcp_create() and svcudp_create() calls create TCP and UDP transport handles (SVCXPRT) respectively, used for receiving and replying to RPC messages. The SVCXPRT transport handle structure is defined in the <RPC\SVC.H> header file.

   If the argument of the svctcp_create() call is RPC_ANYSOCK, the RPC library creates a socket on which to receive and reply to remote procedure calls. The svctcp_create() and clnttcp_create() calls cause the RPC library calls to bind the appropriate socket, if it is not already bound.

   If the argument of the svctcp_create() call is not RPC_ANYSOCK, the svctcp_create() call expects its argument to be a valid socket number. If you specify your own socket, it can be bound or unbound. If it is bound to a port by you, the port numbers of the svctcp_create() and clnttcp_create() calls must match.

   If the send and receive buffer size parameter of svctcp_create() is 0, the system selects a reasonable default.

2. Register the rmtprog service with Portmapper.

   If the rmtprog service terminated abnormally the last time it was used, the pmap_unset() call erases any trace of it before restarting. The pmap_unset() call erases the entry for RMTPROGNUM from the Portmapper's table.

   A service can register its port number with the local Portmapper service by specifying a nonzero protocol number in the svc_register() call. A programmer at the client machine can determine the server port number by consulting Portmapper at the server machine. You can do this automatically by specifying 0 as the port number in the clntudp_create() or clnttcp_create() calls.

   Finally, the program and version number are associated with the rmtprog procedure. The final argument to the svc_register() call is the protocol being used, which in this case is IPPROTO_TCP. Register at the program level, not at the procedure level.

3. Run the remote program RMTPROG.

   The rmtprog service routine must call and dispatch the appropriate XDR calls based on the procedure number. Unlike the registerrpc() call, which performs them automatically, the rmtprog routine requires two tasks:

   - When the NULLPROC procedure (currently 0) returns with no results, use it as a simple test for detecting whether a remote program is running.

   - Check for incorrect procedure numbers. If you detect one, call the svcerr_noproc() call to handle the error.

   As an example, the procedure STRINGPROC has an argument for a character string and returns the character string back to the client. The svc_getargs() call takes an SVCXPRT handle, the xdr_wrapstring() call, and a pointer that indicates where to place the input.

   The user service (rmtprog) serializes the results and returns them to the RPC caller through the svc_sendreply() call.

   Parameters of the svc_sendreply() call include the:

   - SVCXPRT handle
   - XDR routine, which indicates return data type
   - Pointer to the data to be returned

Client Side Program

The following is an example of the lowest layer of RPC on the client side program:

```
#define RMTPROGNUM  (u_long)0x3fffffffL
#define RMTPROGVER  (u_long)0x1L
#define STRINGPROC  (u_long)2

#include <stdio.h>
#include <rpc\rpc.h>
#include <sys\socket.h>
#include <netdb.h>

main(argc, argv)
int argc;
char *argv[ ];
{
   struct hostent *hp;
   struct timeval pertry_timeout, total_timeout;
   struct sockaddr_in server_addr;
   int sock = RPC_ANYSOCK;
   static char buf[100], *strc_in= "Hello", *strc_out=buf;
   char *parrc_in, *parrc_out;
   register CLIENT *clnt;
   enum clnt_stat cs;
...
   /* get the Internet address of RPC server host */
   if ((hp = gethostbyname("RPCSERVER_HOST")) == NULL)
    {
     fprintf(stderr,"Can't get address for %s\n","RPCSERVER_HOST");
     exit (-1);
    }

   pertry_timeout.tv_sec = 3;
   pertry_timeout.tv_usec = 0;

   /* set sockaddr_in structure */
   bcopy(hp->h_addr, (caddr_t)&server_addr.sin_addr.s_addr,
                              hp->h_length);
   server_addr.sin_family = AF_INET;
   server_addr.sin_port = 0;

   /* create clnt TCP handle */
   if ((clnt = clnttcp_create(&server_addr, RMTPROGNUM, RMTPROGVER,
                             &sock, 1024*10, 1024*10)) == NULL)
    {
     clnt_pcreateerror("clnttcp_create fail"); /* Why failed ? */
     exit(-1);
    }

/*
 *  create clnt UDP handle
 *  if ((clnt = clntudp_create(&server_addr, RMTPROGNUM, RMTPROGVER,
 *                              pertry_timeout, &sock)) == NULL)
 *   {
 *     clnt_pcreateerror("clntudp_create fail");
 *     exit(-1);
 *   }
 */
   total_timeout.tv_sec = 10;
   total_timeout.tv_usec = 0;
...

   /*call the remote procedure STRINGPROC associated with */
   /*client handle (clnt) */
   cs=clnt_call(clnt, STRINGPROC,xdr_wrapstring,
   (char *)&strc_in[j],
                    xdr_wrapstring, (char *)&strc_out,total_timeout);
         if (cs != RPC_SUCCESS)
               printf("*Error* clnt_call fail :\n");
   clnt_destroy(clnt);  /* deallocate any memory associated  */
                        /* with clnt handle                  */
...
}
```

The following steps describe the lowest layer of RPC on the client side program:

1. Determine the internet address of the RPC server host.

   Use the gethostbyname() call to determine the internet address of the host, which is running the RPC server. Initialize the socaddr_in structure, found in the <NETINET\IN.H> header file.

   If you are not familiar with socket calls, see Sockets General Programming Information.

2. Use the client RPC handle.

   The clnttcp_create() and clntudp_create() calls create TCP and UDP client RPC handles (CLIENT), respectively. The CLIENT structure is defined in the <RPC\CLNT.H> header file.

   There are six parameters for the clnttcp_create() call:

   - Server address
   - Program number
   - Version number
   - Pointer to a valid socket descriptor
   - Send buffer size
   - Receive buffer size

   Use the same parameters for the clntudp_create() call, except for the send and receive buffer size. Instead, specify a timeout value (between tries).

3. Call the remote procedure.

   The low-level version of the callrpc() call is the clnt_call(), which has seven parameters:

   - CLIENT pointer
   - Remote procedure number (STRINGPROC)
   - XDR call for serializing the argument
   - Pointer to the argument
   - XDR call for deserializing the return value from the RPC server
   - Pointer to where the return value is to be placed
   - Total time in seconds to wait for a reply

   For UDP transport, the number of tries is the clnt_call() timeout divided by the clntudp_create() timeout.

   The return code RPC_SUCCESS indicates a successful call; otherwise, an error has occurred. You find the RPC error code in the <RPC\CLNT.H> header file.

   The clnt_destroy() call always deallocates the space associated with the client handle. If the RPC library opened the socket associated with the client handle, the clnt_destroy() call closes it. If you open the socket, it stays open.

----

rpcgen Command

Use the **rpcgen** command to generate C code to implement an RPC protocol. The input to RPCGEN is a language similar to C, known as RPC language.

You normally use **rpcgen** *infile* to generate the following four output files. For example, if the *infile* is named PROTO.X, **rpcgen** generates:

- A header file called PROTO.H
- XDR routines called PROTOX.C
- Server-side stubs called PROTOS.C
- Client-side stubs called PROTOC.C

For more information on the **rpcgen** command, see the Sun Microsystems publication, *Networking on the Sun Workstation: Remote Procedure Call Programming Guide.*

Syntax

```
  rpcgen   infile                                              `



  rpcgen        -c                                             `
                -h        -o outfile
                -l                          infile
                -m



  rpcgen      -s transport                                     `
                                 -o outfile        infile
```

Parameters

-c Compiles into XDR routines.

-h Compiles into C data definitions (a header file).

-l Compiles into client-side stubs.

-m Compiles into server-side stubs without generating a main routine.

-o *outfile*
Specifies the name of the output file. If none is specified, standard output is used for -c, -h, -l, -m, and -s modes.

infile
Specifies the name of the input file written in the RPC language.

-s *transport*
Compiles into server-side stubs, using the given transport.

----

rpcinfo Command

The **rpcinfo** command makes an RPC call to the RPC server and reports the status of the server, which is registered and operational with Portmapper.

Syntax

rpcinfo for a Host

```
                          local_host
  rpcinfo    -p                                                 `
                          host              > filename
```

Using UDP to send rpcinfo for a Host

```
  rpcinfo                          -u host prognum
               -n portnum                               versnum
                                                                `

       > filename
```

Using TCP to send rpcinfo for a Host

```
    rpcinfo                         -t host prognum
                  -n portnum                          versnum
                                                                          `
          > filename
```

Using UDP to send rpcinfo Broadcast to Hosts

```
    rpcinfo             prognum    versnum                                `
                -b                                > filename
```

Parameters

-p *host*
: Queries the Portmapper about the specified host and prints a list of all registered RPC programs. If the host is not specified, the system defaults to the local host name.

> *filename*
: Specifies a file to which the list of registered RPC programs is redirected.

-n *portnum*
: Specifies the port number to be used for the -t and -u parameters. This value replaces the port number that is given by the Portmapper.

-u *host prognum versnum*
: Uses UDP to send an RPC call to procedure 0 of *prognum* and *versnum* on the specified host, and reports whether a response is received.

-t *host prognum versnum*
: Uses TCP to send an RPC call to procedure 0 of *prognum* and *versnum* on the specified host and reports whether a response is received.

-b *prognum versnum*
: Uses UDP to sends an RPC broadcast to procedure 0 of the specified *prognum* and *versnum* and reports all hosts that respond.

The *prognum* argument can be either a name or a number. If you specify a *versnum*, the **rpcinfo** command tries to call that version of the specified program. Otherwise, it tries to find all the registered version numbers for the program you specify by calling version 0; then it tries to call each registered version.

The TCPIP\ETC\RPC file is associated with the **rpcinfo** command. This file contains a list of server names and their corresponding RPC program numbers and aliases.

Examples

Use the **rpcinfo** command as follows to display RPC services registered on the local host:

```
rpcinfo -p
```

Examples

Use the **rpcinfo** command as follows to display RPC services registered on a remote host named `charm`:

```
rpcinfo -p charm
```

Examples

Use the **rpcinfo** command as follows to display the status of a particular RPC program on the remote host named `charm`:

```
rpcinfo -u charm 100003 2
```

or

```
rpcinfo -u charm nfs 2
```

In the previous examples, the **rpcinfo** command shows one of the following:

```
Program 100003 Version 2 ready and waiting
```

or

```
Program 100003 Version 2 is not available
```

Examples

Use the **rpcinfo** command as follows to display all hosts on the local network that are running a certain version of a specific RPC server:

```
rpcinfo -b 100003 2
```

or

```
rpcinfo -b nfsprog 2
```

In these examples, the **rpcinfo** command lists all hosts that are running Version 2 of the NFS daemon.

Note: The version number is required for the -b parameter.

The enum clnt_stat Structure

The enum clnt_stat structure is defined in the <RPC\CLNT.H> file. RPCs frequently return enum clnt_stat information. The format of the enum clnt_stat structure follows:

```
enum clnt_stat  {
RPC_SUCCESS=0,            /* call succeeded */
/*
 * local errors
 */
RPC_CANTENCODEARGS=1,     /* can't encode arguments */
RPC_CANTDECODERES=2,      /* can't decode results */
RPC_CANTSEND=3,           /* failure in sending call */
RPC_CANTRECV=4,           /* failure in receiving result */
RPC_TIMEDOUT=5,           /* call timed out */
/*
 * remote errors
 */
RPC_VERSMISMATCH=6,       /* RPC versions not compatible */
RPC_AUTHERROR=7,          /* authentication error */
RPC_PROGUNAVAIL=8,        /* program not available */
RPC_PROGVERSMISMATCH=9,   /* program version mismatched */
RPC_PROCUNAVAIL=10,       /* procedure unavailable */
RPC_CANTDECODEARGS=11,    /* decode arguments error */
RPC_SYSTEMERROR=12,       /* generic "other problem" */
/*
 * callrpc errors
 */
RPC_UNKNOWNHOST=13,       /* unknown host name */
/*
 * create errors
 */
RPC_PMAPFAILURE=14,        /* the pmapper failed in its call */
RPC_PROGNOTREGISTERED=15,  /* remote program is not registered */
/*
 * unspecified error
 */
RPC_FAILED=16
            };
```

The Remote Procedure Call Library

To use the RPCs described in this section, you must have the following header files in your H\INCLUDE directory:

RPC Header File	**What It Contains**
RPC\AUTH.H	Authentication interface
RPC\AUTH_UNI.H	Protocol for UNIX-style authentication parameters for RPC
RPC\CLNT.H	Client-side remote procedure call interface
RPC\PMAP_CLN.H	Supplies C routines to get to PORTMAP services
RPC\PMAP_PRO.H	Protocol for the local binder service, or pmap
RPC\RPC.H	Includes the RPC header files necessary to do remote procedure calling
RPC\RPC_MSG.H	Message definitions
RPC\RPCNETDB.H	Data definitions for network utility calls
RPC\RPCTYPES.H	RPC additions to <TYPES.H>
RPC\SVC.H	Server-side remote procedure call interface
RPC\SVC_AUTH.H	Service side of RPC authentication
RPC\XDR.H	eXternal Data Representation serialization routines

The RPC routines are in the RPC32DLL.LIB file in the LIB directory.

Put the following statement at the beginning of any file using RPC code:

```
#include <rpc\rpc.h>
```

Differences between OS/2 and Sun Microsystems RPCs

The IBM OS/2 RPC implementation differs from the Sun Microsystems RPC implementation as follows:

- The global variables *svc_socks[ ]* and *noregistered* are used in place of the *svc_fds* global variable. See svc_socks[ ] for the use of these variables.
- Functions that rely on file descriptor structures are not supported.
- The svc_getreq() call supports the *socks* and *noavail* global variables. In the Sun Microsystems implementation, the svc_getreq() call supports the *rdfds* global variable.
- TYPES.H for RPC has been renamed to RPCTYPES.H.

Compiling an RPC API Application

Follow these steps to compile and link the RPC API application using an IBM 32-bit compiler for OS/2:

1. To compile your program, enter:

   ```
   icc /C myprog.c
   ```

2. To create an executable program, you can enter:

   For VisualAge C++

   ```
   ilink /NOFREEFORMAT myprog,myprog.exe,NULL, rpc32dll.lib
   ```

Notes:

1. The RPC API is not re-entrant. If you are using a multithreaded program, you must serialize the access to the APIs.
2. For more information about the compile and link options, and dynamic link libraries, see the User's Guide provided with your compiler.

File Transfer Protocol

The File Transfer Protocol (FTP) API allows applications to have a client interface for file transfer. Applications written to this interface can communicate with multiple FTP servers at the same time. The interface supports a maximum of 256 simultaneous connections and enables third-party proxy transfers between pairs of FTP servers. Consecutive third-party transfers are allowed between any sequence of pairs of FTP servers.

The FTP API tracks the servers to which an application is currently connected. When a new request for FTP service is requested, the API checks whether a connection to the server exists and establishes one if it does not exist. If the server has dropped the connection since last use, the API re-establishes it.

Note: The FTP API is **not re-entrant**. If you are using a multithreaded program, you must serialize the access to the APIs. For example, without serialization, the program may fail if it has two threads running concurrently and each thread has its own connection to a server.

FTP API Call Library

To use the FTP API, you must have the <FTPAPI.H> header file in your TCPIP\INCLUDE directory. The FTP API routines are in the FTPAPI.LIB file in the LIB directory.

Put the following statement at the top of any file using FTP API code:

```
#include <ftpapi.h>
```

Compiling and Linking an FTP API Application

Follow these steps to compile and link the FTP API application using an IBM 32-bit compiler for OS/2:

1. To compile your program, enter:

```
icc /C myprog.c
```

2. To create an executable program, you can enter:

   For VisualAge C++

   ```
   ilink /NOFREEFORMAT myprog,myprog.exe,NULL,
   ftpapi.lib
   ```

Notes:

1. The FTP API is not re-entrant. If you are using a multithreaded program, you must serialize the access to the APIs.
2. For more information about the compile and link options, and dynamic link libraries, see the User's Guide provided with your compiler.
3. The FTP API can connect to an FTP server using a specific port rather than the well-known port. Code the port number as part of the host name specification, such as `ftpget ("server1 1234", ...)` to connect to server1 by port 1234.

Resource ReSerVation Protocol

The sender and receiver of a data stream use the RSVP (Resource ReSerVation Protocol) to ensure that some quality of service can be reserved on the network. This is in contrast to the usual "best effort" service that is provided, where packets are handled "first come first served."

This section describes the resource reservation protocol implemented in TCP/IP for OS/2.

RSVP Introduction

RSVP is a Quality of Service (QoS) setup protocol. Quality of Service is a set of communication characteristics required by an application. RSVP does not send or receive data. Other protocols do the actual transmitting and receiving of data. A 5-tuple (protocol, destination address, destination port, sender address, sender port) defines the data flow for a QoS. RSVP sets up QoS for flow in a single direction. If two programs will be both sending and receiving and need QoS for each direction, QoS for each flow must be reserved independently. Initial implementations of RSVP have been mainly concerned with UDP unicast or UDP multicast, or other IP protocols, such as TCP.

The RSVPD.EXE daemon implements the RSVP protocol. Applications use an API interface (LIBRSVP.DLL) to communicate with the daemon. The associated toolkit contains a header file (RSVPRAPI.H) and the library stub for the DLL (LIBRSVP.LIB).

The sender initiates a session and causes RSVP path messages to go to the receiver. The path messages indicate what flow the sender is willing to send. RSVP daemons between the sender and receiver can also add information about what flows they can support. A reservation is set up when the receiver sends a reservation to the sender of a path message. The receiver can ask for confirmation that the reservation was set up.

Once a flow is established, the daemons along that flow maintain it automatically by periodically resending path packets from the sender daemon and resending reservation packets in the reverse direction from the receiver's daemon.

There are two types of QoS specification: a controlled load flowspec and a guaranteed flowspec.

Generally, requesting a Quality of Service means at least that a specific bandwidth (number of bytes per second) is requested. There are five parameters in a controlled load flowspec. There are implications in these parameters about how much the packets can get "bunched up." For example, if you send a 100K burst of packets once every two seconds, the average data rate is 50K per second. This data flow could be unacceptable, because the sender or receiver (or intervening routers) might only be able to handle 10K bursts every 0.2 seconds, for a much more even flow of data. In the part of the TCP/IP stack that performs QoS processing on the local machine, setting up for a particular reservation usually involves at least two major actions:

- prioritizing packets as they wait to leave the machine onto the network (or to applications running on the machine)

- preallocation of buffers so that space resources will be adequate for the QoS.

Some applications, like audio data for a conversation, where excessive delay can easily become intolerable, require packets to be delivered within a specified time. Such applications should use the guaranteed flowspec. Besides the five parameters of the controlled load flowspec, two other parameters, guaranteed rate and slack term, specify the delay through the network.

Consequences of Partial RSVP Deployment on a Network

RSVP works in a network where not every node along a data flow path has RSVP implemented. The QoS of transmission through sections of the network can be unpredictable where RSVP is not implemented. The setup process will still be done where the flow passes through RSVP-capable nodes.

Using the RSVP API

This section outlines typical sender and receiver scenarios for using the RSVP API calls.

An RSVP sender session might have these steps:

1. Determine the sender and destination addresses and ports.
2. Start a session with rapi_session(), and provide it with the name of your callback function.
3. Call rapi_getfd() to obtain an alert socket.
4. Call rapi_sender() to establish the program as a sender.
5. Call select() to wait for a read on the alert socket.
6. When data is ready to be read on the alert socket, call rapi_dispatch() to read it.
7. Your callback routine is called as needed by rapi_dispatch().
8. When your callback sees a reservation message from a receiver, go to the next step, otherwise go back to step 5.
9. Process the flowspec information to determine packet size, and so on, and start sending data on a data socket.
10. The select() call on the alert socket should continue to be used to watch for asynchronous error conditions.
11. When you are finished sending the data, call rapi_release() to end the RSVP session.

An RSVP receiver session might have these steps:

1. Determine the sender and destination addresses and ports, and if the destination is a multicast group, join it.
2. Call rapi_session() to start a session, and provide it with your callback function.
3. Call rapi_getfd() to get the alert socket.
4. Use select() to wait for a read on the alert socket.
5. When data is ready to be read on the alert socket, call rapi_dispatch() to read it.
6. Call rapi_dispatch() as needed to call your callback routine.
7. When your callback sees a path message from a sender, go to the next step, otherwise go back to step 4.
8. Process the adspec information in the path message to determine what the reservation should be, and call rapi_reserve() to let the sender(s) see reservation messages.
9. Call select() to begin listening for the data stream on a data socket. Continue to call select() on the alert socket as well, to watch for asynchronous error conditions.

10. When you are finished receiving the data, use rapi_release() to end the RSVP session.

Topics

Determining Addresses and Ports
Starting a Session
Getting an Alert Socket
Establishing the Program as a Sender
Using Select() with the Alert Socket
Callback Function Example
Making a QoS Reservation
Receiving a Reservation Message
Watching for Asynchronous RSVP Events
Closing the RSVP Session

Determining Addresses and Ports

The destination address and port are a specific IP address and port if the data flow is unicast. If the data flow is by way of a multicast group, the destination address and port are for a multicast group. Generally, address and port determination depends upon the protocol to be used to send and receive the data stream.

If the data flow is for the TCP protocol, the usual listen() - connect() - accept() sequence can be done by server and client. Then the IP address and port for each end of the connection are available using the getsockname() and getpeername() calls.

If the data flow is for UDP unicast or multicast, agreement about IP addresses and ports may have to be done externally to the RSVP protocol or the UDP protocol. This would depend upon the application. For example, it might be necessary ahead of time to agree upon a multicast address and port to use for a video broadcast.

When joining a multicast group as a receiver, it is possible to see path messages to that group which are coming from senders. Then a receiver can make reservations for data flows from the senders. A path message includes the IP address and port of the sender.

RSVP is designed to use a variety of network address types. Thus the API uses the more general sockaddr structure. To operate with Internet Protocol addresses, the sockaddr structure must be cast to sockaddr_in.

Starting a Session

The sender and receiver start sessions in the same way, by issuing a rapi_session() call. This call requires a sockaddr structure, which defines a destination address and port, a protocol number, an optional callback function that you provide, and some other parameters. If the session is multicast, then the address is for a multicast group to which the senders send data. The port can be considered an extension of the multicast address. The address and port must agree for all users of that multicast group. The rapi_session() call returns a session ID that is used in subsequent calls to the RSVP API.

This example assumes that the destination is a multicast group.

```
#include
    int retcode;
    rapi_sid_t sessID;          /* RSVP session ID */
    int proto = IPPROTO_UDP;    /* protocol (UDP) */
    struct sockaddr_in mulAddr; /* multicast address, port, protocol */
    int alertSoc;               /* alert socket for asynchronous events */
    int ttl;                    /* multicast time to live value */

    mulAddr.sin_family = AF_INET;
    mulAddr.sin_addr.s_addr = inet_addr("224.1.1.1");
    mulAddr.sin_port = htons(1201);

    /* Do multicast group setup at this point. Code omitted here. */
    sessID = rapi_session((struct sockaddr *)&mulAddr, proto, 0, callback,
                          NULL, &retcode);
    if (! sessID)
        {
        printf("Session did not start! rapi_session() error code %d\n",
               retcode);
        exit(1);
        }
    else
        {
        alertSoc = rapi_getfd(sessID);
        printf("Session %d started, alert socket is %d\n",
               sessID, alertSoc);
        }
    if (alertSoc <= 0)
        {
        printf("Error, rapi_getfd() could not provide an alert socket!\n");
        exit(1);
        }
```

Getting an Alert Socket

The sender and receiver both would issue a rapi_getfd() call at this point. The call returns an alert socket. The socket is used to send data to the RSVP API. The program should issue a select() call on the socket as a read socket in order to know when asynchronous events are available. The program should not read the socket.

If the program is going to receive data, this is a good point to use select() to wait for path messages that indicate that one or more senders are available. There is an example of this in a later section.

If the program is going to send data, this is a good time to tell the RSVP API that the program will be a sender.

Establishing the Program as a Sender

The rapi_sender() call tells RSVP that the program will be a sender. This call establishes the sender address and port for the data flow, and it specifies the characteristics of the data stream in RSVP terms, with a tspec (transmission specification). This information will go to the destination in an RSVP path message. The RSVP API will take care of repeating the message as needed according to the RSVP protocol.

This example shows the use of rapi_sender():

```
int retcode;
    struct sockaddr_in senderAddr; /* sender address, port, protocol */
    rapi_tspec_t sndTSpec;

    /* set up senderAddr and sndTSpec */

    senderAddr.sin_family = AF_INET;
    senderAddr.sin_addr.s_addr = inet_addr("129.5.24.1");
    senderAddr.sin_port = htons(1024);


/* set the object size and form in the object header */
sndTSpec.len = sizeof(sndTSpec);
sndTSpec.form = RAPI_FORMAT_IS_GEN;
/* fill in the body of the Tspec object */
    sndTSpec.tspec_r = 100000;
    sndTSpec.tspec_b = 2600;
    sndTSpec.tspec_p = 100000;
    sndTSpec.tspec_m = 1300;
    sndTSpec.tspec_M = 1300;

    retcode = rapi_sender(
            sessID,           /* Session ID                              */
            0,                /* Flags - not defined                       */
            (struct sockaddr *)&senderAddr,
                              /* Local host: (Src addr, port), net order */
            NULL,             /* Sender template - not supported         */
            &sndTSpec,        /* Sender tspec                            */
            NULL,             /* Sender adspec - not supported           */
            NULL,             /* Sender policy data - not supported      */
            ttl               /* TTL of multicast session (if multicast) */
            );
    if (retcode)
        printf("rapi_sender() error %d\n", retcode);
```

Using Select() with the Alert Socket

The alert socket is used to communicate messages from the RSVP daemon to the API. These messages generally turn into events for the program. The select() call is used to wait for read data on the alert socket. Once there is read data available, the program must call rapi_dispatch() so that the RSVP API can read the data and handle it properly. Typically, rapi_dispatch() will, in turn, call your callback function, possibly more than once.

In the <RSVPRAPI.H> header file, the data type rapi_event_rtn_t is declared to be a pointer to a callback function. The arguments of the function are also declared there, so that you can see how to declare your callback function arguments.

This example shows a loop with a select() call waiting on the alert socket until a path message is received and handled by a callback function:

```
fd_set readSockets;
    int rc, rapi_rc;
    int receivedPathEvent = 0; /* set to 1 by the callback routine */

    while (! receivedPathEvent)
        {
        FD_ZERO(&readSockets);
        FD_SET(alertSoc, &readSockets);
        if ((rc = select(FD_SETSIZE, &readSockets, NULL, NULL, &timeout)) < 0)
            {
            psock_errno("select() on alert socket");
            exit(1);
            }
        if (rc > 0 && FD_ISSET(alertSoc, &readSockets))
            {
            rapi_rc = rapi_dispatch();
            if (rapi_rc == RAPI_ERR_NORSVP)
                {
                printf("Warning! RSVP has gone away.\n");
                exit(1);
                }
            }
        } /* end while */
```

Waiting for a reservation message would be handled similarly.

Callback Function Example

This example mainly shows the code that works with the previous example, which looks for a path message. This example also prints all the event information, using the format routines of the RSVP API where appropriate:

```
struct sockaddr_in sndAddr; /* remember the sender IP addr:port here */

    int _System callback(
        rapi_sid_t         sid,          /* Which sid generated event    */
        rapi_eventinfo_t   eventType,    /* Event type                  */
        rapi_styleid_t     styleID,      /* Style ID                    */
        int                errorCode,    /* Error code                  */
        int                errorValue,   /* Error value                 */
        struct sockaddr   *pNodeAddr,    /* Error node address          */
        u_char             errorFlags,   /* Error flags                 */
        int                nFilterSpecs, /* Number of filterspecs/sender*/
                                         /*      templates in list      */
        rapi_filter_t     *pFilterSpec,  /* Filterspec/sender templ list*/
        int                nFlowSpecs,   /* Number of flowspecs/Tspecs  */
        rapi_flowspec_t   *pFlowSpec,    /* Flowspec/tspec list         */
        int                nAdSpecs,     /* Number of adspecs           */
        rapi_adspec_t     *pAdSpec,      /* Adspec list                 */
        void *             pClientArg    /* Client supplied arg         */
        )
    {
    int i;
    #define FMT_BUF_SIZE 600
    char buf[FMT_BUF_SIZE];
    printf("callback() sid %d, eventType %d, styleID %d\n",
           sid, eventType, styleID);

    /* if we received the path event then tell the select loop */
    if (eventType == RAPI_PATH_EVENT)
        {
        receivedPathEvent = 1;
        /* get the sender address from pFilterSpec */
        if (nFilterSpec && (pFilterSpec->form == RAPI_FILTERFORM_BASE))
           sndAddr = pFilterSpec->filter.base.sender;
        /* else it is another address form... */
        }
```

In a realistic program, the adspec information would be processed to determine what services are supported by network elements that support RSVP. Information on data rate, bandwidth, packet MTU, and so on, normally would be available. A receiver would determine what reservation flowspec would be suitable for the data stream that the sender could send.

If a reservation event is being processed by a sender, the adspec information normally would provide the upper limit on packet size, and other useful information, and the sender could adjust its data flow accordingly.

To print all the event information:

```
    if (errorCode == RAPI_ERR_OK)
        printf("errorCode %d\n", errorCode);
    else
        printf("errorCode %d, errorValue %d, nodeAddr %s:%d, errorFlags %d\n",
               errorCode, errorValue,
               inet_ntoa(((struct sockaddr_in*)pNodeAddr->sin_addr),
               ((struct sockaddr_in*)pNodeAddr->sin_port, errorFlags);
    if (nFilterSpecs)
        for (i = 0; i < nFilterSpecs; ++i)
            {
            rapi_fmt_filtspec(pFilterSpec + i, buf, FMT_BUF_SIZE);
            printf("filterspec %d, %s\n", i, buf);
            }
    else
        printf("No filter specs\n");
    if (nFlowSpecs)
        for (i = 0; i < nFlowSpecs; ++i)
            {
            rapi_fmt_flowspec(pFlowSpec + i, buf, FMT_BUF_SIZE);
            printf("flowspec %d, %s\n", i, buf);
            }
    else
        printf("No flowspecs\n");
    if (nAdSpecs)
        for (i = 0; i < nAdSpecs; ++i)
            {
            rapi_fmt_adspec(pAdSpec + i, buf, FMT_BUF_SIZE);
            printf("adspec %d, %s\n", i, buf);
            }
    else
        printf("No adspecs\n");
    /* the function must return a value, but the API does nothing to it */
    return 0;
    } /* end callback */
```

Making a QoS Reservation

The rapi_reserve() call is used by the receiver to make a reservation or change a reservation. The call specifies a reservation style, filterspecs (data senders), and flowspecs (QoS specifications).

This example shows how a receiver makes or changes a reservation:

```
#define MAX_RSVP_SENDERS 10
    int retcode;
    rapi_styleid_t rapiStyle = RAPI_RSTYLE_FIXED; /* fixed reservation */
    int filterSpecCount;
    rapi_filter_t filterSpec[MAX_RSVP_SENDERS];
    int flowSpecCount;
    rapi_flowspec_t flowSpec[MAX_RSVP_SENDERS];

    filterSpecCount = 1;   /* one specified sender this time */
    /* set the object size and form in the object header */
    filterSpec[0].len = sizeof(rapi_hdr_t) + sizeof(rapi_filter_base_t);
    filterSpec[0].form = RAPI_FILTERFORM_BASE;
    /* fill in the body of the filterspec object */
    filterSpec[0].filter.base.sender = sndAddr; /* copied from path event */

    flowSpecCount = 1;     /* one flowspec this time */
    /* set the object size and form in the object header */
    flowSpec[0].len = sizeof(rapi_hdr_t) + sizeof(CL_flowspec_t);
    flowSpec[0].form = RAPI_FORMAT_IS_CL;
    /* fill in the body of the flowspec object */
    flowSpec[0].cl_tspec_r = 100000;
    flowSpec[0].cl_tspec_b = 2600;
    flowSpec[0].cl_tspec_p = 100000;
    flowSpec[0].cl_tspec_m = 1300;
    flowSpec[0].cl_tspec_M = 1300;

    retcode = rapi_reserve(
            sessID,  /* session id */
            RAPI_REQ_CONFIRM,    /* flags */
            NULL,    /* rcv host addr (optional, sessID has destination) */
            rapiStyle,    /* style ID */
            NULL,    /* style extension, not supported */
            NULL,    /* receiver policy, not supported */
            filterSpecCount,
            filterSpec,   /* array of filterspecs */
            flowSpecCount,
            flowSpec      /* array of flowspecs */
            );
    if (retcode)
        printf("rapi_reserve() error %d\n", retcode);
```

The rapi_reserve() call above has the optional RAPI_REQ_CONFIRM flag, that asks for a confirmation message to be sent if the reservation is made. Such an event indicates that the reservation had a very high probability of succeeding.

After making the reservation, the receiver should start listening for the data stream on a data socket.

Receiving a Reservation Message

After a sender's callback function receives a reservation message, the sender can start sending data to the destination. The RSVP data flow has been set up with the reserved Quality of Service.

Start listening for the data stream on a data socket.

Watching for Asynchronous RSVP Events

Both a sender and a receiver should continue to use a select() call on the alert socket to watch for asynchronous error conditions. The select() call can be in separate thread, or it can be a select() call that is used in sending or receiving the data stream.

Closing the RSVP Session

Use the rapi_release() call to close an RSVP session. When the program ends, the RSVP API will also close any sessions that the program has open.

Reservation Styles

Bandwidth reservation relates to the way the bandwidth is to be allocated (the reservation type), and the technique used to select senders (the sender selection). The reservation type may be distinct for individual data flows, or may be shared among multiple data flows. The selection of the senders may be explicit (meaning that individual senders are identified individually), or it may be a wildcard selection that implicitly selects all the senders to a session. This table shows the reservation styles that are defined for the various Sender Selection-Reservation Type combinations:

Sender Selection	Reservation Type	Reservation Style
explicit	distinct	Fixed-Filter (FF)
explicit	shared	Shared-Explicit (SE)
wildcard	distinct	(none defined)
wildcard	shared	Wildcard-Filter (WF)

The fixed filter (FF) style specifies a distinct flow for each sender. The bandwidth reservations are made separately for each flow. Parameters in the reservation specify each sender explicitly.

With the shared explicit (SE) style, each of the senders is specified explicitly in the reservation message, but the reserved bandwidth is shared by all the senders wherever they can be merged upstream from the receiver.

The wildcard filter (WF) style specifies a single reservation of bandwidth which is to be shared by all the senders. This type of reservation is propagated upstream to all senders as they become senders in the RSVP session. Besides there being a single bandwidth reservation for all the senders to the receiver that makes the reservation, the other receivers in the same multicast group will have their bandwidth reservations merged as well. The "largest" such reservation for the session at any point in the multicast tree determines how much bandwidth is reserved at that point in the tree.

The wildcard and shared explicit styles are primarily useful for multicast applications where the data sources are unlikely to transmit simultaneously.

Tspecs and Flowspecs

The sender tspec contains a structure with five fields that define the flow. The two receiver flowspecs also contain this structure.

```
    typedef struct {
       float32_t  IS_Tspec_r; /* Token bucket rate (IP packet bytes/sec) */
```

```
      float32_t  IS_Tspec_b; /* Token bucket depth (bytes) */
      float32_t  IS_Tspec_p; /* Peak data rate (IP packet bytes/sec) */
      u_int32_t  IS_Tspec_m; /* Min Policed Unit (bytes) */
      u_int32_t  IS_Tspec_M; /* Max packet size (bytes) */
   }   IS_Tspec_t;
```

The token bucket rate (*r*) is in bytes of IP datagrams per second. That is, the packets must include the IP packet header (20 bytes) and either the UDP header (8 bytes) or the TCP header (20 bytes), as well as the data in the packet, when computing packet sizes. The valid value of *r* may range from 1 byte per second to 40 terabytes per second. The bucket depth (*b*) may range from 1 byte to 250 gigabytes.

The maximum packet size (*M*) should be no bigger than the maximum transmission unit (MTU) of the path between sender and receiver. The value of m must be no bigger than *M*.

In a tspec the sender indicates, for any arbitrary time period of *T* seconds, that the amount of data sent in that period will not be greater than $r*T+b$. When computing this, any packet that is less than the minimum policed unit (m) in size should be counted as *m* bytes.

The token bucket model is a way of viewing the transmission of data. The sender has a stream of data that comes from some source, and the sender expects to send it using RSVP. The sender may need to send at a specified rate dictated by the data stream source, for example an audio or video data stream. The receiver may need to output the data in a timely fashion after receiving it as well. The token bucket rate is the rate at which the sender will send. Imagine bytes (tokens) filling the token bucket. When enough accumulate so that the above formula is not violated, a packet is built and sent. The bucket is reduced by the amount that was sent.

The peak data rate (*p*) is mainly used in guaranteed flowspecs. It can be set the same as r, for controlled load flowspecs. In guaranteed flowspecs, it is required that for any time period *T*, that the amount of data sent in that period cannot exceed $M+p*T$.

A Guaranteed flowspec has a tspec and an rspec. An rspec is shown below.

```
   typedef struct {
      float32_t  Guar_Rspec_R; /* Guaranteed Rate (IP packet bytes/sec) */
      u_int32_t  Guar_Rspec_S; /* Slack term (microseconds) */
   }   Guar_Rspec_t;
```

The guaranteed rate (*R*) and slack term (*S*) are computed by the receiver based upon the values in the adspec in the path message. They are explained in a later section.

A sender and receiver in an RSVP session negotiate reasonable values of packet size, and so forth, by using information in the path and reservation messages. Typically, a sender specifies the largest packet it is willing to send. This value will be in the path message tspec. When the path message is seen by a receiver, the packet size (in the adspec) will be the maximum that can be handled by all the RSVP-enabled nodes that the message passed through. (If one or more nodes was not RSVP-enabled, then that fact is also indicated in the adspec. In that case, the reservation that is made should be considered unreliable.) Once the receiver makes a reservation, if the reservation is subsequently merged with other reservations then a similar process can happen as the reservation message goes toward the sender. Thus the sender can see what the maximum packet size should be for the packets to be sent reliably to all receivers. If a packet is larger than that size, it will be sent by "best effort" technique rather than according to the reservation that was made.

A similar process occurs for delay through the network, so that the receiver has an estimate of what sort of service it can expect and request.

Adspecs

Adspecs are a part of path events. Adspecs carry information about the RSVP support that is possible in the network for a specified flow.

An adspec is created by the RSVP daemon when a sender calls rapi_sender(). The path message traverses the network to the receiver, and accumulates changes to the adspec when passing through each RSVP enabled network node. The receiver callback function is called with a path event. The path event has the original sender tspec, and the adspec that indicates the support that is possible through the network for that tspec. The receiver then calls rapi_reserve() to make a reservation based upon the information in the tspec and adspec. Then the reservation message traverses the network to the sender. The QoS reservation is set up in each RSVP node until the reservation message reaches the sender. The sender callback function is called with a reserve event. The sender then may adjust packet size or other factors in order to use the reservation effectively, based upon the reservation information.

When the application callback function is passed a non-NULL pointer for the adspec, then the pointer points to an IS_adspec_t data type:

```
   typedef struct {
           Gen_adspec_t    Gen_Adspec;
           rbool_t         CL_present;
```

```
        CL_adspec_t     CL_Adspec;
        rbool_t         Guar_present;
        Guar_adspec_t   Guar_Adspec;
    } IS_adspec_t;
```

The IS_adspec_t data type has several fields. The Gen_Adspec field is always present. If CL_present is true, then CL_Adspec contains a CL_adspec_t data type, which is a composite of the controlled flow support for the path. If Guar_present is true, then Guar_Adspec contains a Guar_adspec_t data type, which is a composite of the guaranteed flow support for the path.

The Gen_Adspec_t data type is shown below.

```
    typedef struct {
        rbool_t    servBreak;    /* break in service */
        u_int16_t  ISHops;       /* num Int-serv aware hops */
        float32_t  pathBW;       /* min path band width (bytes/sec) */
        u_int32_t  pathLatency;  /* min path latency (microseconds) */
        u_int16_t  composedMTU;  /* composed path MTU */
    } Gen_adspec_t;
```

If servBreak is true, then there is at least one hop on the path that does not have QoS support and QoS must be considered unreliable. ISHops specifies the number of hops that do have QoS support of some sort (controlled load or guaranteed). There are estimates for minimum path bandwidth and minimum delay through the path. The composedMTU is the maximum transmission unit that can be sent on the path without fragmentation of the packet. The composedMTU includes the IP packet header and whatever protocol headers (UDP, TCP, and so on) will be in the packet. Note that a particular type of service might have a different value for the composedMTU, due to implementation differences for various types of QoS.

The CL_adspec_t data type provides information specifically for controlled load flows on the path. Note that it is possible for there to be a break in service for controlled load even if QoS is supported at every node. Some nodes may only implement QoS for guaranteed flows.

```
    typedef struct {
        rbool_t    servBreak;    /* break in service */
        u_int16_t  ISHops;       /* num Int-serv aware hops */
        float32_t  pathBW;       /* min path band width (bytes/sec) */
        u_int32_t  pathLatency;  /* min path latency (microseconds) */
        u_int16_t  composedMTU;  /* composed path MTU */
    } CL_adspec_t;
```

The fields have the same significance as for Gen_adspec_t but refer only to path hops for which controlled load is supported.

The Guar_adspec_t data type provides information specifically for guaranteed flows on the path. Note that it is possible for there to be a break in service for guaranteed flows even if QoS is supported at every node. Some nodes may only implement controlled load QoS.

```
    typedef struct {
        rbool_t    servBreak;    /* break in service */
        u_int16_t  ISHops;       /* num Int-serv aware hops */
        float32_t  pathBW;       /* min path band width (bytes/sec) */
        u_int32_t  pathLatency;  /* min path latency (microseconds) */
        u_int16_t  composedMTU;  /* composed path MTU (bytes) */
        u_int32_t  Ctot;         /* total rate dep. err (bytes) */
        u_int32_t  Dtot;         /* total rate indep. err (microseconds) */
        u_int32_t  Csum;         /* reshaped rate dep. err (bytes) */
        u_int32_t  Dsum;         /* reshaped rate indep. err (microseconds) */
    } Guar_adspec_t;
```

The first five fields have the same significance as for Gen_adspec_t but refer only to path hops for which guaranteed flow is supported. The remaining fields are unique to guaranteed flows.

Controlled Load Reservations

The controlled load reservation requests a reservation for a guaranteed data rate. The receiver is asking for an IP packet byte rate that is no greater than $r*T+b$ bytes for any arbitrary time of T seconds. The receiver expects to not need more than b bytes of packet space for arriving packets in the TCP/IP stack, while it is busy processing previously received packets. The receiver should request a maximum packet size M no bigger than the composedMTU from the controlled load adspec. When computing $r*T+b$, packets smaller than m should be treated as if they were size m. The peak rate p in the reservation is checked for validity but usually is ignored.

For more information, refer to Internet RFC 1633, *Integrated Services in the Internet Architecture: an Overview* , and Internet Draft *draft-ietf-intserv-ctrl-load-svc-07* .

Guaranteed Reservations

Guaranteed reservations are more complex than controlled load because guaranteed flow provides a way to reserve a specified maximum delay through the network, as well as a specified data rate. Guaranteed service does not control the average or minimum packet delay. Guaranteed service also does not control jitter, the difference between minimum and maximum packet delays.

The guaranteed flow model of the network assumes that there are two sources of delay in the network, the delay (*D*) that is independent of rate of flow, and the delay (*C*) that is proportional to rate of flow. Delay for a rate *r* would be `C*r+D` in an ideal situation. In practice, the RSVP-enabled network elements provide an upper bound for their contribution to *C* and *D* , and a lower bound for *D* . The lower bound for *C* is 0. The receiver chooses a reservation that takes these bounds into consideration.

The pathLatency field of the adspec for guaranteed service is an estimate of the minimum delay through the network for that type of service. It can be estimated from many diverse factors such as speed of light through fiber cables or minimum path length of routing code. Each RSVP-enabled node adds its own estimate of this value to the adspec.

The delay that is based on the data rate can include many factors, such as time to queue a packet, and maximum time to wait for the current packet to be sent before the next one can be sent from an interface.

When a path message arrives at the receiver, the guaranteed flow adspec will contain Ctot and Dtot, which are accumulations of the C and D delay factors on the path through the network. The terms Csum and Dsum are used by intermediate nodes along with the slack term that the guaranteed flow will provide. These terms are used by nodes that will "reshape" the data flow to make it conform to the flow parameters.

These are the formula variables and their sources, for determining the Guaranteed flowspec.

Var	Spec	Spec Field	Description
Bmin	Guar_adspec_t	pathBW	Minimum path bandwidth (IP packet bytes per second)
MTU	Guar_adspec_t	composedMTU	Composed path MTU (bytes)
Ctot	Guar_adspec_t	Ctot	Total rate-dependent error (bytes)
Dtot	Guar_adspec_t	Dtot	Total rate-independent error (microseconds)
r	IS_Tspec_t	IS_Tspec_r	Token bucket rate (IP packet bytes per second)
b	IS_Tspec_t	IS_Tspec_b	Token bucket depth (bytes)
p	IS_Tspec_t	IS_Tspec_p	Peak data rate (IP packet bytes per second)
m	IS_Tspec_t	IS_Tspec_m	Minimum Policed Unit (bytes)
M	IS_Tspec_t	IS_Tspec_M	Maximum packet size (bytes)
R	Guar_Rspec_t	Guar_Rspec_R	Guaranteed rate (IP packet bytes per second)
S	Guar_Rspec_t	Guar_Rspec_S	Slack term (microseconds)

Values for *r* , *b* , *p* , *m* , *M* , *R* , and *S* are to be computed for the guaranteed reservation. Assume that the reservation *r* , *b* , *p* , and *m* are chosen based on the original tspec values in the path message. If *Bmin* is greater than *r* or *p* , they should be increased at least to *Bmin* . Assume that the reservation *M* is set to a reasonable value for the application that is not greater than MTU. Assume that the minimum delay is not greater than the maximum that the application can tolerate. Note that all computation described below should be floating point , especially the division by 1000000 that converts between microseconds and seconds.

If *r* <= *p* <= *R* then the upper bound on end-to-end requested delay (*Dreq*) in seconds is:

```
Dreq = (M+Ctot)/R + Dtot/1000000
```

If *r* <= *R* < *p* then the upper bound on *Dreq* in seconds is

```
Dreq = [(b-M)/R*(p-R)/(p-r)] + (M+Ctot)/R + Dtot/1000000
```

The higher the value of *R* is, the lower *Dreq* will be. The application will choose an *R* value that is sufficiently high that the maximum delay is sufficient for what the sender and receiver need to do. Note that studies have shown that packets will often arrive much more quickly than the maximum delay, and therefore the application must be prepared to buffer them.

If the peak rate *p* is unknown or unspecified, it should be set to infinity. Infinity is represented in IEEE floating point with an exponent of all one bits (255), and a sign and mantissa of all zero bits. In that case the upper bound on end-to-end delay in seconds simplifies to:

```
Dreq = b/R + Ctot/R + Dtot/1000000
```

The slack term *S* is the difference between the requested maximum delay and the desired delay, and must be a non-negative number. A typical desired delay is when *R* is set to *r*, in the ideal fluid model of flow. The delay in that model is

```
Dideal = b/r + Ctot/r + Dtot/1000000
```

The slack term can then be:

```
S = Dreq - Dideal.
```

If the application chooses an *S* greater than zero, then RSVP-enabled nodes on the path can use it and the *Csum* and *Dsum* values to adjust their local reservations to lower the amount of resources allocated to the flow.

Note that a guaranteed reservation may have its values *R* and *S* adjusted by intervening nodes, so that the reservation seen by the sender in the reservation message may be different from what the receiver provided with rapi_reserve() call.

For more information, refer to Internet RFC 1633, *Integrated Services in the Internet Architecture: an Overview* , and Internet Draft *draft-ietf-intserv-ctrl-load-svc-07* .

Reference Information

This section describes:

- Protocol-Independent C Sockets API

  Describes the protocol-independent socket calls supported by networking services. This information includes call syntax, usage, and related information.

- TCP/IP Network Utility Routines API

  Describes the sockets utility and Sockets Secure Support (SOCKS) function calls supported by networking services. This information includes call syntax, usage, and related information.

- Remote Procedure and eXternal Data Representation API

  Describes the remote procedure and XDR function calls along with their syntax, usage, and related information.

- File Transfer Protocol API

  Describes the file transfer protocol function calls along with their syntax, usage, and related information.

- Resource ReSerVation Protocol API

Describes the resource reservation protocol function calls along with their syntax, usage, and related information.

Protocol-Independent C Sockets API

The following table briefly describes each protocol-independent socket call supported by networking services and identifies where you can find the syntax, parameters, and other appropriate information. The socket calls described in this section can be used to access services for all protocols.

Note: If you are using the internet communications domain (PF_INET protocol family), you can use all APIs in the following table and those in TCP/IP Network Utility Routines API.

Protocol-Independent Sockets API Quick Reference

Socket Call	Description
accept()	Accepts a connection request from a remote host
accept_and_recv()	Accepts a connection on a socket, receives the first message from the connected client, and returns the local and peer addresses
addsockettolist()	Adds a socket to the list of owned sockets for the calling process
bind()	Binds a local name to the socket
connect()	Requests a connection to a remote host
getpeername()	Gets the name of the peer connected to socket
getsockname()	Gets the local socket name
getsockopt()	Gets the socket options associated with a socket
ioctl()	Performs special operations on sockets
listen()	Completes the binding necessary for a socket to accept connections and creates a connection request queue for incoming requests
os2_ioctl()	Performs special operations on sockets; in particular, operations related to returning status from kernel
os2_select()	Gets read, write, and exception status on a group of sockets (OS/2 version)
psock_errno()	Writes a short error message to the standard error device
readv()	Receives data on a socket into a set of buffers
recv()	Receives data on a connected socket
recvfrom()	Receives data on a socket
recvmsg()	Receives data and control information on a socket

removesocketfromlist()	Removes a socket from the list of owned sockets for the calling process
select()	Gets read, write, and exception status on a group of sockets (BSD version)
send()	Sends data on a connected socket
send_file()	Sends the file data over a connected socket
sendmsg()	Sends data and control information on a socket
sendto()	Sends data on a socket
setsockopt()	Sets options associated with a socket
shutdown()	Shuts down all or part of a full duplex connection
so_cancel()	Cancels a pending blocking sockets API call on a socket
sock_errno()	Returns error code set by a socket call
socket()	Creates an endpoint for communication and returns a socket descriptor representing the endpoint
soclose()	Shuts down a socket and frees resources allocated to the socket
sysctl()	Performs special operations on the TCP/IP stack kernel
writev()	Writes data from a set of specified buffers on a socket

----

accept()

The accept() socket call accepts a connection request from a remote host. Raccept() accepts a connection request from a SOCKS server. See Socket Secure Support for information about SOCKS.

Syntax

```
#include <types.h>
#include <sys\socket.h>
#include <netinet\in.h>
#include <netnb\nb.h>
#include <sys\un.h>
int accept(s, name, namelen)
int s;
sockaddr *name;
int *namelen;
```

Parameters

s

Socket descriptor.

name
: Pointer to a sockaddr structure that contains the socket address of the connection client when the accept() call returns. The format of *name* is determined by the communications domain where the client resides. This parameter can be NULL if the caller is not interested in the client address.

namelen
: Must initially point to an integer that contains the size in bytes of the storage pointed to by *name*. On return, that integer contains the size of the data returned in the storage pointed to by *name*. If *name* is NULL, *namelen* is ignored and can be NULL.

Description

This call is used by a server acting in a connection-oriented mode to accept a connection request from a client. The call accepts the first connection on its queue of pending connection requests. The accept() call creates a new socket descriptor with the same properties as *s* and returns it to the caller. The new socket descriptor cannot be used to accept new connections. The original socket, *s*, remains available to accept more connection requests.

If the queue has no pending connection requests, accept() blocks the caller unless *s* is in nonblocking mode. If no connection requests are queued and *s* is in nonblocking mode, accept() returns a value of -1 and sets the return code to SOCEWOULDBLOCK.

The *s* parameter must be a socket descriptor created with the socket() call. It is usually bound to an address with the bind() call and must be made capable of accepting connections with the listen() call. The listen() call marks the socket as one that accepts connections and allocates a queue to hold pending connection requests. The listen() call allows the caller to place an upper boundary on the size of the queue.

The *name* parameter is a pointer to a buffer where the connection requester address is placed. The *name* parameter is optional and can be set to be the NULL pointer. If set to NULL, the requester address is not copied into the buffer. The exact format of *name* depends on the communications domain where the communication request originated. For example, if the connection request originated in the internet domain, *name* points to a sockaddr_in structure as defined in the header file <NETINET\IN.H>.

The *namelen* parameter is used only if *name* is not NULL. Before calling accept(), you must set the integer pointed to by *namelen* to the size, in bytes, of the buffer pointed to by *name*. On successful return, the integer pointed to by *namelen* contains the actual number of bytes copied into the buffer. If the buffer is not large enough to hold the address, up to *namelen* bytes of the requester address are copied.

This call is used only with SOCK_STREAM or SOCK_SEQPACKET sockets. You cannot screen requesters without calling accept(). The application cannot tell the system the requesters it will accept connections from. The caller can, however, choose to close a connection immediately after discovering the identity of the requester.

The select() call can be used to check the socket for incoming connection requests.

Return Values

A non-negative socket descriptor indicates success; the value -1 indicates an error. You can get the specific error code by calling sock_errno() or psock_errno().

Error Code	**Description**
SOCENOTSOCK	The *s* parameter is not a valid socket descriptor.
SOCEFAULT	Using *name* and *namelen* would result in an attempt to copy the address into a portion of the caller address space into which information cannot be written.
SOCEINTR	Interrupted system call.
SOCEINVAL	Listen() was not called for socket *s*.
SOCENOBUFS	Insufficient buffer space available to create the new socket.
SOCEOPNOTSUPP	The *s* parameter is not connection-oriented.
SOCEWOULDBLOCK	The *s* parameter is in nonblocking mode and no connections are on the queue.
SOCECONNABORTED	The software caused a connection close.

Examples

The following are two examples of the accept() call. In the first, the caller wants to have the requester address returned. In the second, the caller does not want to have the requester address returned.

```
int clientsocket;
int s;
struct sockaddr clientaddress;
int addrlen;
int accept(int s, struct sockaddr *addr, int *addrlen); /* extracted from sys/socket.h */
```

```
/* socket(), bind(), and listen() have been called */
/* EXAMPLE 1: I want the address now */
addrlen = sizeof(clientaddress);
clientsocket = accept(s, &clientaddress, &addrlen);
/* EXAMPLE 2: I can get the address later using getpeername() */
clientsocket = accept(s, (struct sockaddr *) 0, (int *) 0);
```

Related Calls

- accept_and_recv()
- bind()
- connect()
- getpeername()
- getsockname()
- listen()
- sock_errno()
- socket()

----

accept_and_recv()

The accept_and_recv( ) API accepts a connection on a socket, receives the first message from the connected client, and returns the local and peer addresses.

Syntax

```
#include <types.h>
#include <sys\socket.h>
#include <netinet\in.h>
#include <sys\time.h>
int accpet_and_recv(s,&sock_accex,&cli_addr,&cli_len,&local,&locallen,outbuff,alloc)
int s;
long sock_accex;
struct sockaddr_in cli_addr;
long cli_len;
struct sockaddr_in local;
long locallen;
char * outbuff;
int alloc;
```

Parameters

s
: Socket descriptor of the listening socket.

sock_accex
: Pointer to an int that specifies the socket on which to accept the connection. This pointer should be initialized to -1 so that the kernel accepts the socket and returns this pointer to the application using this parameter.

cli_addr
: Pointer to a sockaddr structure where the address of the connecting socket will be returned.

cli_len
: Pointer to a socklen_t that, on output, specifies the length of the stored address.

local_addr
: Pointer to a sockaddr structure where the address of the connecting socket will be returned.

locallen
: Pointer to a socklen_t that, on, output specifies the length of the stored address.

outbuff
: Pointer to the buffer where the message should be stored.

alloc
Length in bytes of the buffer pointed to by the buffer argument.

Description

The accept_and_recv( ) call combines the socket functions accept( ) and recv( ) into a single API transition. The accept_and_recv( ) function accepts a new connection, receives the first block of data from the client and returns the local and remote addresses to the application. The thread sleeping on accept_and_recv( ) wakes-up only after it gets the first data block from the client.

Return Values

The total number of bytes received in the receive buffer associated with the accept_and_recv() is returned upon successful completion and a value of -1 is returned in case of an error.

Error Code	**Description**
EBADF	The sockfd or the sock_accex is not a valid descriptor.
ECONNABORTED	A connection has been ended.
ECONNRESET	A connection was forcibly closed by a peer.
EFAULT	The buffer pointed to by sock_accex,cli_addr, clilen, local, locallen or buffer was not valid.
EISCONN	The sock_accex is either bound or already connected.
ENOTSOCK	The sockfd or the sock_accex argument does not refer to a socket.
EOPNOTSUPP	The socket type of the specified socket does not support accepting connections or the O_NONBLOCK is set for this socket and non-blocking is not supported for this function, or the accept_and_recv( ) function is not supported by this version of TCP/IP.
ENOREUSE	Socket reuse is not supported.
EINTR	The accept_and_recv( ) function was interrupted by a signal that was caught before a valid connection arrived.
EINTRNODATA	The accept_and_recv( ) function was interrupted by a signal that was caught after a valid connection arrived but before the first block of data.
EINVAL	The sockfd is not accepting connections.
EMFILE	{OPEN_MAX} descriptors are currently open in the calling process.
ENFILE	The maximum number of descriptors in the system are already open.
EIO	An I/O error occured.
ENOBUFS	No buffer space available.
ENOMEM	There was insufficient memory available to complete the operation.
EPROTO	A protocol error occured.
ENOSR	There are insufficient STREAMS resources available for the operation to complete.

Related Calls

accept()
recv()
bind()
connect()
getpeername()
getsockname()
listen()
sock_errno()
socket()

----

addsockettolist()

The addsockettolist() call adds a socket to the list of owned sockets for the calling process.

Syntax

```
#include <types.h>
#include <sys\socket.h>
void addsockettolist(s)
int s;
```

Parameters

s
  Socket descriptor.

Description

When a process ends, the sockets library automatically cleans up sockets by registering an exit list handler. This exit routine closes all open sockets that are maintained in a process's socket list. When a process is initiated the list is empty, and whenever a socket(), accept(), or soclose() call is made the list is updated. The addsockettolist() call provides a mechanism to transfer socket ownership to another process. The addsockettolist() call adds the socket indicated by the *s* parameter to the calling process's socket ownership list.

Return Values

The value 1 indicates success; the value 0 indicates an error.

Related Calls

removesocketfromlist()

----

bind()

The bind() socket call binds a local name to the socket. Rbind() binds a SOCKS local name to the socket. See Socket Secure Support for information about SOCKS.

Syntax

```
#include <types.h>
#include <sys\socket.h>
int bind(s, name, namelen)
int s;
struct sockaddr *name;
int namelen;
```

Parameters

s
  Socket descriptor returned by a previous call to socket().

name
  Pointer to a sockaddr structure containing the name that is to be bound to *s*.

namelen
  Size in bytes of the sockaddr structure pointed to by *name*.

Description

The bind() call binds a unique local name to the socket with descriptor *s*. After calling socket(), a descriptor does not have a name associated with it. However, it does belong to a particular addressing family as specified when socket() is called. The exact format of a name depends on the addressing family. The bind() procedure also allows servers to specify from which network interfaces they wish to receive UDP packets and TCP connection requests.

If *s* was created in the AF_INET domain, the format of the name buffer is expected to be *sockaddr_in* as defined in the header file <NETINET\IN.H>:

```
struct in_addr
{
        u_long s_addr;
};
struct sockaddr_in
{
        u_char  sin_len;
        u_char  sin_family;
        u_short sin_port;
        struct  in_addr sin_addr;
        char    sin_zero[8];
};
```

The *sin_len* field is ignored. The *sin_family* field must be set to AF_INET. The *sin_port* field is set to the port that the application must bind to. It must be specified in network byte order. If *sin_port* is set to 0, the caller leaves it to the system to assign an available port. The application can call getsockname() to discover the port number assigned. The *sin_addr* field is set to the internet address and must be specified in network byte order. On hosts with more than one network interface (called multihomed hosts), a caller can select the interface that the host will bind to.

Subsequently, only UDP packets or TCP connection requests which match the bound name from this interface are routed to the socket. If *sin_addr* is set to the constant INADDR_ANY, as defined in <NETINET\IN.H>, the caller is requesting that the socket be bound to all network interfaces on the host. After this, UDP packets or TCP connections which match the bound name from all interfaces are routed to the socket. This becomes important when a server offers a service to multiple networks. By leaving the address unspecified, the server can accept all UDP packets or TCP connection requests made for its port, regardless of the network interface on which the requests arrived. The *sin_zero* field is not used and must be set to all zeros.

In the NetBIOS (AF_NET) domain, set all 16 characters in *snb_name* in the sockaddr_nb structure to binary zeros (null). The system will generate a name for the socket.

Return Values

The value 0 indicates success; the value -1 indicates an error. You can get the specific error code by calling sock_errno() or psock_errno().

Error Code	**Description**
SOCEADDRINUSE	The address is already in use. See the SO_REUSEADDR option described under getsockopt() and the SO_REUSEADDR option described under setsockopt().
SOCEADDRNOTAVAIL	The address specified is not valid on this host. For example, the internet address does not specify a valid network interface.
SOCEAFNOSUPPORT	The address family is not supported.
SOCENOTSOCK	The *s* parameter is not a valid socket descriptor.
SOCEFAULT	Using *name* and *namelen* would result in an attempt to copy the address into a non-writable portion of the caller's address space.
SOCEINVAL	The socket is already bound to an address, or *namelen* is not the expected length.
SOCENOBUFS	No buffer space is available.

Examples

Note the following about the bind() call examples:

- For the internet examples, put the internet address and port in network-byte order. To put the port into network-byte order, use the htons() utility routine to convert a short integer from host-byte order to network-byte order.
- For the internet examples, set the *address* field using the inet_addr() utility routine, which takes a character string representing the dotted-decimal address of an interface and returns the binary internet address representation in network-byte order.
- Zero the structure before using it to ensure that the name requested does not set any reserved fields.

See connect() for examples of how a client might connect to servers.

```
int rc;
int s;
struct sockaddr_in myname;
```

```
int bind(int s, struct sockaddr *name, int namelen); /* extracted from sys/socket.h */
/* Bind to a specific interface in the internet domain */
/* clear the structure */
memset(&myname, 0, sizeof(myname));
myname.sin_len = sizeof(myname);
myname.sin_family = AF_INET;
myname.sin_addr = inet_addr("129.5.24.1"); /* specific interface */
myname.sin_port = htons(1024);
...
rc = bind(s, (struct sockaddr *) &myname, sizeof(myname));

/* Bind to all internet network interfaces on the system */
/* clear the structure */
memset(&myname, 0, sizeof(myname));
myname.sin_len = sizeof(myname);
myname.sin_family = AF_INET;
myname.sin_addr.s_addr = INADDR_ANY; /* all interfaces */
myname.sin_port = htons(1024);
...
rc = bind(s, (struct sockaddr *) &myname, sizeof(myname));

/* Bind to a specific interface in the internet domain.
Let the system choose a port                         */
/* clear the structure */
memset(&myname, 0, sizeof(myname));
myname.sin_len = sizeof(myname);
myname.sin_family = AF_INET;
myname.sin_addr = inet_addr("129.5.24.1"); /* specific interface */
myname.sin_port = 0;
...
rc = bind(s, (struct sockaddr *) &myname, sizeof(myname));

/* Bind to a unique NetBIOS name on adapter 0 */
struct sockaddr_nb nbname;
memset(&nbname, 0, sizeof(nbname));
nbname.sin_len = sizeof(nbname);
nbname.snb_family = AF_NB;
nbname.snb_type = NB_UNIQUE;
nbname.snb_adapter = 0;
strcpy(nbname.snb_name, "NBSERVER"); /* Note that a NetBIOS name is
                                        16 bytes long.  In this example,
                                        the last 8 bytes are filled
                                        with zeros.                    */

...
rc = bind(s, (struct sockaddr *) &nbname, sizeof(nbname));
```

Related Calls

- connect()
- gethostbyname()
- getsockname()
- htons()
- inet_addr()
- listen()
- sock_errno()
- socket()

----

connect()

The connect() socket call requests a connection to a remote host.

Syntax

```
#include <types.h>
#include <sys\socket.h>
int connect(s, name, namelen)
int s;
```

```
struct sockaddr *name;
int namelen;
```

Parameters

s
: Socket descriptor used to originate the connection request.

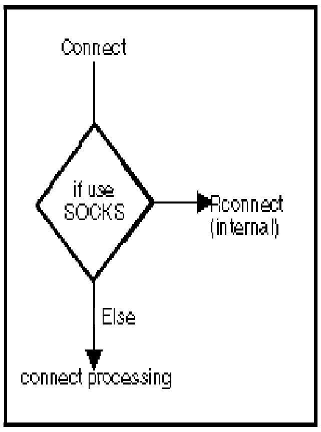
name
: Pointer to a sockaddr structure containing the address of the socket to which a connection will be attempted.

namelen
: Size in bytes of the sockaddr structure pointed to by *name* .

Description

The following diagram illustrates connect() processing:

connect() Processing



If you are using a SOCKS server, connect() calls Rconnect(). See Socket Secure Support for information about SOCKS.

Stream or sequenced packet sockets: The connect() call performs two tasks when called for a stream or sequenced packet socket:

1. Completes the binding if necessary for a socket
2. Attempts to create a connection between two sockets.

This call is used by the client side of socket-based applications to establish a connection with a server. The remote server must have a passive open pending. This means the server must successfully call bind() and listen(); otherwise, connect() returns a value of -1 and the error value is set to SOCECONNREFUSED. If you are using a SOCKS server and the connection is rejected by the SOCKS server, the return code will be SOCECONNREFUSED.

In the internet communication domain, a timeout occurs if a connection to the remote host is not successful within 75 seconds (1 minute and 15 seconds). There is no timeout for Local IPC. In the NetBIOS communication domain, a timeout occurs if a connection to the host is not successful within the time defined by the NetBIOS protocol parameters *Transmit Timer* multiplied by *Transmit Retry* .

If *s* is in blocking mode, the connect() call blocks the caller until the connection is established or until an error is received. If the socket is in nonblocking mode, and the connection was successfully initiated, connect() returns a value of -1 and sets the error value to SOCEINPROGRESS. The caller can test the completion of the connection setup by calling:

- select(), to test for the ability to write to the socket
- getsockopt(), with option SO_ERROR, to test if the connection succeeded

Stream or sequenced packet sockets can call connect() only once.

Datagram or raw sockets: The connect() call specifies the destination peer address when called for a datagram or raw socket. Normally, datagram and raw sockets use connectionless data transfer calls such as sendto() and recvfrom(). However, applications can call connect() to specify and store the destination peer address for this socket. The system will then know which address to send data to on this socket. This method of communication allows datagram and raw sockets to be connected. However, data is still not guaranteed to be delivered. Thus the normal features of connectionless mode sockets are maintained. The address is remembered until another connect() call is made. This permits the use of readv(), recv(), send(), and writev(), which are usually reserved for connection-oriented sockets. The application can still use sendto(), recvfrom(), sendmsg(), and recvmsg(). The advantage of calling connect() and being connected is that the destination peer address does not have to be specified for all datagrams sent.

Datagram and raw sockets can call connect() multiple times. The application can reset their destination address by specifying a new address on the connect() call. In addition, the socket can be returned to operate in a connectionless mode by calling connect() with a null destination address. The null address is created by zeroing the sockaddr structure and only setting the address family field. The call to connect will return a value of -1, indicating that the connection to the null address cannot be established. Calling sock_errno() will return SOCEADDRNOTAVAIL. For more information on connecting datagram sockets, see Description for sendto().

Return Values

The value 0 indicates success; the value -1 indicates an error. You can get the specific error code by calling sock_errno() or psock_errno(). If you are using a SOCKS server and the SOCKS server rejects the connection, the return code will be SOCECONNREFUSED.

Error Code	**Description**
SOCEADDRNOTAVAIL	The calling host cannot reach the specified destination.
SOCEAFNOSUPPORT	The address family is not supported.
SOCEALREADY	The socket *s* is marked nonblocking, and a previous connection attempt has not completed.
SOCENOTSOCK	The *s* parameter is not a valid socket descriptor.
SOCECONNREFUSED	The connection request was rejected by the destination host. If you are using a SOCKS server and the SOCKS server rejects the connection, the return code will be SOCECONNREFUSED.
SOCEFAULT	Using *name* and *namelen* would result in an attempt to copy the address into a portion of the caller's address space to which data cannot be written.
SOCEINPROGRESS	The socket *s* is marked nonblocking, and the connection cannot be completed immediately. The SOCEINPROGRESS value does not indicate an error condition.
SOCEINTR	Interrupted system call.
SOCEINVAL	The *namelen* parameter is not a valid length.
SOCEISCONN	The socket *s* is already connected.
SOCENETUNREACH	The network cannot be reached from this host.
SOCETIMEDOUT	The connection establishment timed out before a connection was made.
SOCENOBUFS	No buffer space is available.
SOCEOPNOTSUPP	The operation is not supported on socket *s*.

Examples

Note the following about these connect() call examples:

- For the internet examples, put the internet address and port in network-byte order. To put the port into network-byte order, use the htons() utility routine to convert a short integer from host-byte order to network-byte order.
- For the internet examples, set the *address* field using the inet_addr() utility routine, which takes a character string representing the dotted-decimal address of an interface and returns the binary internet address representation in network-byte order.
- To ensure that the name requested does not set any reserved fields, zero the structure before using it.

These examples could be used to connect to the servers shown in the examples listed for bind().

```
int s;
struct sockaddr_in servername;
int rc;
int connect(int s, struct sockaddr *name, int namelen); /* extracted from sys/socket.h */
/* Connect to server bound to a specific interface in the internet domain */
/* clear the structure */
memset(&servername, 0, sizeof(servername));
servername.sin_len = sizeof(servername);
servername.sin_family = AF_INET;
servername.sin_addr.s_addr = inet_addr("129.5.24.1"); /* specific interface */
servername.sin_port = htons(1024); /* set to the port to which */
                                   /* the server is bound */
...
rc = connect(s, (struct sockaddr *) &servername, sizeof(servername));

/* Connect to a NetBIOS server */
struct sockaddr_nb nbservername;
memset(&nbservername, 0, sizeof(nbservername));
nbservername.snb_len = sizeof(nbservername);
nbservername.snb_family = AF_NB;
nbservername.snb_type = NB_UNIQUE;
nbservername.snb_adapter = 0;
strcpy(nbservername.snb_name, "NBSERVER");
...
rc = connect(s, (struct sockaddr *) &nbservername, sizeof(nbservername));
```

Related Calls

- accept()
- accept_and_recv()
- bind()
- getsockname()
- htons()
- inet_addr()
- listen()
- Rconnect()
- select()
- send()
- sock_errno()
- socket()

----

getpeername()

The getpeername() socket call gets the name of the peer connected to socket.

Syntax

```
#include <types.h>
#include <sys\socket.h>
int getpeername(s, name, namelen)
int s;
struct sockaddr *name;
int *namelen;
```

Parameters

s
Socket descriptor.

name
Pointer to a sockaddr structure. The name of the peer connected to socket *s* is returned. The exact format of *name* is determined by the domain where communication occurs.

namelen
Pointer to the size in bytes of the sockaddr structure pointed to by *name* .

Description

This call returns the name of the peer connected to socket *s* . The *namelen* parameter must be initialized to indicate the size of the space pointed to by *name* . On return, *namelen* is set to the size of the peer name copied. If the buffer is too small, the peer name is truncated.

The getpeername() call operates only on connected sockets. If the connection is through a SOCKS server, the address returned will be that of the SOCKS server.

A process can use the getsockname() call to retrieve the local address of a socket.

Return Values

The value 0 indicates success; the value -1 indicates an error. You can get the specific error code by calling sock_errno() or psock_errno().

Error Code	**Description**
SOCENOTSOCK	The *s* parameter is not a valid socket descriptor.
SOCEFAULT	Using the *name* and *namelen* parameters as specified would result in an attempt to access storage outside of the address space of the caller.
SOCENOTCONN	The socket is not connected.
SOCENOBUFS	No buffer space is available.

Related Calls

- accept()
- bind()
- connect()
- getsockname()
- Rgetsockname()
- sock_errno()
- socket()

----

getsockname()

The getsockname() socket call gets the local socket name. If you are using a SOCKS server, see Socket Secure Support for information about SOCKS.

Syntax

```
#include <types.h>
#include <sys\socket.h>
int getsockname(s, name, namelen)
int s;
struct sockaddr *name;
int *namelen;
```

Parameters

s
: Socket descriptor.

name
: Pointer to a sockaddr structure. The name of *s* is returned.

namelen
: Pointer to the size in bytes of the buffer pointed to by *name* .

Description

This call returns the name for the socket specified by the *s* parameter in the structure pointed to by the *name* parameter. It returns the address to the socket that has been bound. If the socket is not bound to an address, the call returns with the family set and the rest of the structure is set to zero. For example, an unbound socket in the internet domain would cause the name to point to a sockaddr_in structure with the *sin_family* field set to AF_INET and all other fields zeroed.

The *namelen* parameter must be initialized to indicate the size of the space pointed to by *name* and is set to the size of the local name copied. If the buffer is too small, the local name is truncated.

Sockets are explicitly assigned a name after a successful call to bind(). Stream and sequenced packet sockets are implicitly assigned a name after a successful call to connect() or accept() if bind() was not called.

If the socket is connected through a SOCKS server, this call returns the IP address and port of the local machine that is being used to communicate with the SOCKS server.

The getsockname() call is often used to discover the port assigned to a socket after the socket has been implicitly bound to a port. For example, an application can call connect() without previously calling bind(). In this case, the connect() call completes the binding necessary by assigning a port to the socket.

A process can use the getpeername() call to determine the address of a destination socket in a socket connection.

Return Values

The value 0 indicates success; the value -1 indicates an error. You can get the specific error code by calling sock_errno() or psock_errno().

Error Code	Description
SOCENOTSOCK	The *s* parameter is not a valid socket descriptor.
SOCEFAULT	Using the *name* and *namelen* parameters as specified would result in an attempt to access storage outside of the address space of the caller.
SOCENOBUFS	No buffer space available.

Related Calls

- accept()
- accept_and_recv()
- bind()
- connect()
- getpeername()
- Rgetsockname()
- sock_errno()
- socket()

----

getsockopt()

The getsockopt() socket call gets the socket options associated with a socket.

Syntax

```
#include <types.h>
#include <sys\socket.h>
int getsockopt(s, level, optname, optval, optlen)
int s;
int level;
int optname;
char *optval;
int *optlen;
```

Parameters

s
Socket descriptor.

level
Specifies which option level is being queried for the specified *optname* .

optname
Name of a specified socket option. Only one option can be specified on a call.

optval
Pointer to buffer to receive the option data requested.

optlen
Pointer to the size of the buffer.

Description

This call returns the value of a socket option at the socket or protocol level. It can be called for sockets of all domain types. Some options are supported only for specific socket types. You must specify the level of the option and the name of the option to retrieve option values. The following table lists the supported levels.

Supported Levels

Supported Level	#define in
SOL_SOCKET	<SYS\SOCKET.H>
IPPROTO_IP	<NETINET\IN.H>
IPPROTO_TCP	<NETINET\IN.H>
NBPROTO_NB	<NETNB\NB.H>

The *optval* parameter is a pointer to the buffer where the option values are returned. The *optlen* parameter must be initially set to the size of the buffer before calling getsockopt(). On return, the *optlen* parameter is set to the actual size of the data returned. For socket options that are Boolean, the option is enabled if *optval* is nonzero and disabled if *optval* is 0.

The following tables list the supported options for getsockopt() at each level (SOL_SOCKET, IPPROTO_IP, IPPROTO_TCP). Detailed descriptions of the options follow each table.

Supported getsockopt() Socket Options for SOL_SOCKET

Option Name	Description	Domains(*)	Data Type	Boolean or Value
SO_ACCEPTCONN	Listen status	I, L	int	Boolean
SO_BROADCAST	Allow sending of broadcast messages	I, N	int	Boolean
SO_DEBUG	Turn on recording of debugging information	I, L	int	Boolean
SO_DONTROUTE	Bypass routing tables	I, L	int	Boolean
SO_ERROR	Return any pending error and clear	I, L	int	Value
SO_KEEPALIVE	Keep connections alive	I	int	Boolean
SO_LINGER	Linger on close if data present	I	struct linger	Value
SO_L_BROADCAST	Limited broadcast sent on all interfaces	I	int	Boolean
SO_OPTIONS	Retrieve socket options (flags)	I	int	Flags
SO_OOBINLINE	Leave received OOB data in-line	I	int	Boolean
SO_RCVBUF	Receive buffer size	I, L, N	int	Value
SO_RCV_SHUTDOWN	If shutdown called for receive	I, L	int	Boolean
SO_RCVLOWAT	Receive low watermark	I, L	int	Value
SO_RCVTIMEO	Receive timeout	I, L	struct timeval	Value
SO_REUSEADDR	Allow local address reuse	I, N	int	Boolean
SO_REUSEPORT	Allow local port reuse	I	int	Boolean
SO_SNDBUF	Send buffer size	I, L, N	int	Value

SO_SND_SHUTDOWN	If shutdown called for send	I, L	int	Boolean
SO_SNDLOWAT	Send low watermark	I, L	int	Value
SO_SNDTIMEO	Send timeout	I, L	struct timeval	Value
SO_TYPE	Socket type	I, L, N	int	Value
SO_USELOOPBACK	Bypass hardware when possible	I, L	int	Boolean

Table Note (*) This column specifies the communication domains to which this option applies: I for internet, L for local IPC, and N for NetBIOS.

The following options are recognized for SOL_SOCKET:

Option	Description
SO_ACCEPTCONN	Returns true if the socket is in the listen state.
SO_BROADCAST	(Datagram sockets only.) Retrieves the current status of the SO_BROADCAST option. When this option is enabled, the application can send broadcast messages over *s*, if the interface specified in the destination supports broadcasting of packets.
SO_DEBUG	Retrieves the current status of the SO_DEBUG option.
SO_DONTROUTE	Retrieves the current status of the SO_DONTROUTE option. When this option is enabled, it causes outgoing messages to bypass the standard routing algorithm and be directed to the appropriate network interface, according to the network portion of the destination address. When enabled, packets can be sent only to directly connected networks (networks for which this host has an interface).
SO_ERROR	Returns any pending error on the socket and clears the error status. It can be used to check for asynchronous errors on connected datagram sockets or for other asynchronous errors (errors that are not returned explicitly by one of the socket calls).
SO_KEEPALIVE	(Stream sockets only.) Retrieves the current status of the SO_KEEPALIVE option. TCP uses a timer called the keepalive timer. This timer is used to monitor idle connections that might have been disconnected because of a peer crash or timeout. When this option is enabled, a keepalive packet is periodically sent to the peer. This is mainly used to allow servers to close connections that are no longer active as a result of clients going away without properly closing connections.
SO_LINGER	(Stream sockets only.) Retrieves the current status of the SO_LINGER option. When this option is enabled and there is unsent data present when soclose() is called, the calling application is blocked during the soclose() call until the data is transmitted or the connection has timed out. When this option is disabled, the soclose() call returns without blocking the caller, and TCP waits to try to send the data. Although the data transfer is usually successful, it cannot be guaranteed, because TCP waits only a finite amount of time to send the data. The *optval* parameter points to a linger structure, defined in <SYS\SOCKET.H>: **Field** / **Description** *l_onoff* / Option on/off *l_linger* / Linger time The *l_onoff* field is set to zero if the SO_LINGER option is being disabled. A nonzero value enables the option. The *l_linger* field specifies the amount of time in seconds to linger on close. A value of zero will cause soclose() to wait until the disconnect completes.
SO_L_BROADCAST	Gets limited broadcast sent on all interfaces.
SO_OPTIONS	Gets the current socket options from the stack. The socket option flags are returned as a 32-bit variable. The *so_xxx* socket option flags are defined in <SYS\SOCKET.H>.

SO_OOBINLINE	(Stream sockets only.) Retrieves the current status of the SO_OOBINLINE option. When this option is enabled, it causes out-of-band data to be placed in the normal data input queue as it is received, making it available to recv(), and recvfrom() without having to specify the MSG_OOB flag in those calls. When this option is disabled, it causes out-of-band data to be placed in the priority data input queue as it is received, making it available to recv() and recvfrom() only by specifying the MSG_OOB flag in those calls.
SO_RCVBUF	Retrieves buffer size for input. This value tailors the receive buffer size for specific application needs, such as increasing the buffer size for high-volume connections.
SO_RCV_SHUTDOWN	Returns true if shutdown was called for receive.
SO_RCVLOWAT	Retrieves the receive low watermark.
SO_RCVTIMEO	Retrieves the receive timeout. The *optval* parameter is a pointer to a timeval structure, which is defined in <SYS\TIME.H>.
SO_REUSEADDR	(Stream and datagram sockets only.) Retrieves the current status of the SO_REUSEADDR option. When this option is enabled, local addresses that are already in use can be bound. This alters the normal algorithm used in the bind() call. At connect time, the system checks to be sure that no local address and port have the same foreign address and port. The error SOCEADDRINUSE is returned if the association already exists.
SO_REUSEPORT	(Stream and datagram sockets only.) Retrieves the current status of the SO_REUSEPORT option. This option specifies that the rules used in validating ports supplied by a subroutine of the bind() call should allow reuse of a local port/address combination. Each binding of the port/address combination must specify the socket option. This option enables or disables the reuse of local port/address combinations.
SO_SNDBUF	Retrieves the send buffer size. This value tailors the send buffer size for specific application needs, such as increasing the buffer size for high-volume connections.
SO_SND_SHUTDOWN	Returns true if the shutdown function was called as part of the send() call.
SO_SNDLOWAT	Retrieves the send low watermark.
SO_SNDTIMEO	Retrieves the send timeout. *optval* is a pointer to a timeval structure, which is defined in <SYS\TIME.H>.
SO_TYPE	Retrieves the socket type. On return, the integer pointed to by *optval* is set to one of the following: SOCK_STREAM, SOCK_DGRAM, or SOCK_RAW.
SO_USELOOPBACK	Bypasses hardware when possible.

Supported getsockopt() Socket Options for IPPROTO_IP

Option Name	Description	Data Type	Boolean or Value
IP_HDRINCL	Header is included with data	int	Boolean
IP_MULTICAST_IF	Default interface for outgoing multicasts	struct in_addr	Value
IP_MULTICAST_LOOP	Loopback of outgoing multicast	uchar	Boolean
IP_MULTICAST_TTL	Default TTL for outgoing multicast	uchar	Value
IP_OPTIONS	IP options	char *	Value
IP_RECVDSTADDR	Queueing IP destination address	int	Boolean
IP_RECVTRRI	Queueing token ring routing information	int	Boolean
IP_RETOPTS	IP options	char *	Value
IP_TOS	IP type of service for outgoing datagrams	int	Value

```
 IP_TTL              IP time to live for      int            Value
                     outgoing datagrams
```

The following options are recognized for IPPROTO_IP:

Option	**Description**
IP_HDRINCL	(Raw sockets only.) Specifies whether the IP header is included with data.
IP_MULTICAST_IF	Retrieves the default interface for outgoing multicasts.
IP_MULTICAST_LOOP	Retrieves the value of the loopback setting for outgoing multicast.
IP_MULTICAST_TTL	Retrieves the default time to live for outgoing multicast packets.

IP_OPTIONS	Retrieves IP options. Same as IP_RETOPTS.
	The data type is char * ip_retopts[4], such as
	`ip_retopts[0]=IPOPT_OPTVAL`
	`ip_retopts[1]=IPOPT_OLEN`
	`ip_retopts[2]=IPOPT_OFFSET`
	`ip_retopts[3]=IPOPT_MINOFF`
	For an example that uses IP_RETOPTS, see Example of IP_RETOPTS Socket Call.

IP_RECVDSTADDR	(UDP only.) Retrieves queueing IP destination address. This option must get this information through a recvmsg() call. For more information, see Example of recvmsg() Call.
IP_RECVTRRI	(UDP packets on token ring only.) Retrieves the flag that indicates whether queueing of token ring routing information is enabled. This TTRI information must be received as control data through a recvmsg() call. For more information , see Example of recvmsg() Call.

IP_RETOPTS	Retrieves IP options to be included in outgoing datagrams.
	The data type is char * ip_retopts[4], such as
	`ip_retopts[0]=IPOPT_OPTVAL`
	`ip_retopts[1]=IPOPT_OLEN`
	`ip_retopts[2]=IPOPT_OFFSET`
	`ip_retopts[3]=IPOPT_MINOFF`
	For an example that uses IP_RETOPTS, see Example of IP_RETOPTS Socket Call.

IP_TOS	Retrieves IP type of service for outgoing datagrams.
IP_TTL	Retrieves IP time to live value for outgoing datagrams.

Supported getsockopt() Socket Options for IPPROTO_TCP

```
 Option Name        Description              Data Type     Boolean
                                                           or Value

 TCP_CC             Connection count function int          Boolean

 TCP_MAXSEG         Maximum segment size     int           Value

 TCP_MSL            TCP MSL value            int           Value

 TCP_NODELAY        Don't delay send to      int           Boolean
                    coalesce packets

 TCP_TIMESTMP       Time stamp function      int           Boolean

 TCP_WINSCALE       Window scale function    int           Boolean
```

The following options are recognized for IPPROTO_TCP:

Option	**Description**
TCP_CC	(T/TCP only.) Retrieves the connection count function enabled/disabled status flag (RFC 1644). For more information about T/TCP, see TCP Extensions for Transactions (T/TCP).
TCP_MAXSEG	Retrieves the maximum segment size.
TCP_MSL	Retrieves the TCP Maximum Segment Lifetime (MSL) value.
TCP_NODELAY	(Stream sockets only.) Retrieves the current status of the TCP_NODELAY option. Disables the buffering algorithm so that the client's TCP sends small packets as soon as possible. This often has no performance effects on LANs, but can degrade performance on WANs.
TCP_TIMESTMP	(T/TCP only.) Retrieves the TCP timestamp function enabled/disabled status flag (RFC 1323). For more information about high performance, see TCP Extensions for High Performance (RFC 1323).
TCP_WINSCALE	(T/TCP only) Retrieves the window scale function enabled/disabled status flag (RFC 1323). For more information about high performance, see TCP Extensions for High Performance (RFC 1323).

Supported getsockopt() Socket Options for NBPROTO_NB

```
 Option Name       Description              Data Type     Boolean
                                                          or Value

 NB_DGRAM_TYPE     Type of datagrams to     int           Value
                   receive
```

The following option is recognized for NBPROTO_NB:

Option	**Description**	
NB_DGRAM_TYPE	(Datagram sockets only.) Retrieves the type of datagrams to be received on the socket. The possible values are:	
	NB_DGRAM	The socket is to receive normal (unicast) datagrams only.
	NB_BROADCAST	The socket is to receive broadcast datagrams only.
	NB_DGRAM_ANY	The socket can receive both normal or broadcast datagrams.

Return Values

The value 0 indicates success; the value -1 indicates an error. You can get the specific error code by calling sock_errno() or psock_errno().

sock_errno() Value	**Description**
SOCEADDRINUSE	The address is already in use.
SOCENOTSOCK	The *s* parameter is not a valid socket descriptor.
SOCEFAULT	Using *optval* and *optlen* parameters would result in an attempt to access memory outside the caller's address space.
SOCENOPROTOOPT	The *optname* parameter or *level* parameter is not recognized.

Examples

The following are examples of the getsockopt() call. See setsockopt() for examples of how the options are set.

```
int rc;
int s;
int optval;
int optlen;
struct linger lstruct;
```

```
int getsockopt(int s, int level, int optname, char *optval, int *optlen);
/* extracted from sys/socket.h */
...
/* Is out of band data in the normal input queue? */
optlen = sizeof(int);
rc = getsockopt( s, SOL_SOCKET, SO_OOBINLINE, (char *) &optval, &optlen);
if (rc == 0)
{
    if (optlen == sizeof(int))
    {
         if (optval)
            /* yes it is in the normal queue */
         else
            /* no it is not                  */
    }
}
...
/* Do I linger on close? */
optlen = sizeof(lstruct);
rc = getsockopt( s, SOL_SOCKET, SO_LINGER, (char *) &lstruct, &optlen);
if (rc == 0)
{
    if (optlen == sizeof(lstruct))
    {
        if (lstruct.l_onoff)
           /* yes I linger */
        else
           /* no I do not  */
    }
}
```

The following is an example of the ip_retopts socket option.

Example of IP_RETOPTS Socket Call

```
/* [0]:IPOPT_OPTVAL, [1]:IPOPT_OLEN, [2]:IPOPT_OFFSET, [3]:IPOPT_MINOFF */
char ip_retopts[8];

main()
{
    int optlen, sraw, i;


    if ((sraw = socket(AF_INET, SOCK_RAW, IPPROTO_RAW)) < 0)
        psock_errno("Socket()");

    printf("IP_RETOPTS or OP_OPTIONS will get/set the IP options \n");
    ip_retopts[IPOPT_OPTVAL] = IPOPT_TS ;    /* TimeStamp IP options to set */
    ip_retopts[IPOPT_OLEN]   = 8;
    ip_retopts[IPOPT_OFFSET] = 4;
    ip_retopts[IPOPT_MINOFF] = 4;

    printf("Setting the IP_RETOPTS to TimeStamp option (%d) \n",
                      ip_retopts[IPOPT_OPTVAL]);
    if (setsockopt(sraw,IPPROTO_IP,IP_RETOPTS,(char *)&ip_retopts[0] ,
                      sizeof(ip_retopts)) < 0)
        psock_errno("setsockopt() IP_RETOPTS");

    /* NOTE ::: when the getsockopt returns it will stick in the first hop    */
    /*          destination in the first 4 bytes by shifting all data right.  */
    memset(ip_retopts, 0, sizeof(ip_retopts));
    printf("Get the ip_retopts value set for this socket\n");
    optlen = sizeof(ip_retopts);
    if (getsockopt(sraw,IPPROTO_IP,IP_OPTIONS,(char *)ip_retopts,&optlen) < 0) {
        psock_errno("getsockopt() IP_RETOPTS ");
    }
    else {
       if  (ip_retopts[4+IPOPT_OPTVAL] == IPOPT_TS)
         printf ("IP_RETOPTS now set to TimeStamp option(%d) \n",
                    ip_retopts[4+IPOPT_OPTVAL]);
       else
         printf ("IP_RETOPTS now set to ??? (%d) \n",
                    ip_retopts[4+IPOPT_OPTVAL]);
    }
    soclose(sraw);
}
```

Related Calls

- bind()
- endprotoent()
- getprotobyname()
- getprotobynumber()
- getprotoent()
- setprotoent()
- setsockopt()
- sock_errno()
- socket()

----

ioctl()

The ioctl() socket call performs special operations on sockets.

Syntax

```
#include <types.h>
#include <sys\socket.h>
#include <sys\ioctl.h>
#include <net\route.h>
#include <net\if.h>
#include <net\if_arp.h>
int ioctl(s, cmd, data)
int s;
int cmd;
caddr_t data;
```

Parameters

s
: Socket descriptor.

cmd
: Command to perform.

data
: Pointer to the data associated with *cmd*.

Description

This call controls the operating characteristics of sockets. The *data* parameter is a pointer to data associated with the particular command, and its format depends on the command that is requested.

Option	Description
FIOASYNC	This option has no effect.
FIONBIO	Sets or clears nonblocking input/output for a socket. When this option is set, input/output calls will not block until the call is completed. The *data* parameter is a pointer to an integer. If the integer is 0, nonblocking input/output on the socket is cleared. Otherwise, the socket is set for nonblocking input/output.
FIONREAD	Gets the number of immediately readable bytes for the socket. The *data* parameter is a pointer to an integer. Sets the value of the integer to the number of immediately readable characters for the socket.

Internet: The following ioctl commands are supported for the internet domain:

Option	**Description**
OSIOCGIFADDR	Provided for compatibility with releases of TCP/IP prior to 4.21. The *data* parameter is a pointer to an ifreq structure. The interface address is returned in the old sockaddr format in the argument.
OSIOCGIFDSTADDR	Provided for compatibility with releases of TCP/IP prior to 4.21. The *data* parameter is a pointer to an ifreq structure. The destination address is returned in the old sockaddr format in the argument.
OSIOCGIFBRDADDR	Provided for compatibility with releases of TCP/IP prior to 4.21. The *data* parameter is a pointer to an ifreq structure. The broadcast address is returned in the old sockaddr format in the argument.
OSIOCGIFCONF	Provided for compatibility with releases of TCP/IP prior to 4.21. The *data* parameter is a pointer to an ifreq structure. The interface configuration is returned in the old sockaddr format in the argument.
OSIOCGIFNETMASK	Provided for compatibility with releases of TCP/IP prior to 4.21. The *data* parameter is a pointer to an ifreq structure. The interface netmask is returned in the old sockaddr format in the argument.
SIOCADDMULTI	Adds a 48-bit physical multicast address. This works only for Ethernet. The *data* parameter is a pointer to an ifreq structure.
SIOCADDRT	Adds a routing table entry. The *data* parameter is a pointer to an rtentry structure, as defined in <NET\ROUTE.H>. The routing table entry, passed as an argument, is added to the routing tables.
SIOCAIFADDR	Adds an IP address for an interface. The *data* parameter is a pointer to an ifaliasreq structure, which is defined in <NET\IF.H>
SIOCARP	Sends an ARP request to all interfaces for a given IP address. The *data* parameter is a pointer to the IP address in the type of an unsigned long integer.
SIOCATMARK	Queries whether the current location in the data input is pointing to out-of-band data. The *data* parameter is a pointer to an integer. Sets the argument to 1 if the socket points to a mark in the data stream for out-of-band data. Otherwise, sets the argument to 0.
SIOCDARP	Deletes an arp table entry. The *data* parameter is a pointer to an *arpreq* as defined in <NET\IF_ARP.H>. The arp table entry passed as an argument is deleted from the arp tables, if it exists.
SIOCDELMULTI	(Ethernet only.) Deletes a 48-bit physical multicast address. The *data* parameter is a pointer to an ifreq structure, which is defined in <NET\IF.H>.
SIOCDELRT	Deletes a routing table entry. The *data* parameter is a pointer to a rtentry structure, as defined in <NET\ROUTE.H>. If it exists, the routing table entry passed as an argument is deleted from the routing tables.
SIOCDIFADDR	Deletes an IP address for an interface. The *data* parameter is a pointer to an ifreq structure, defined in <NET\IF.H>.
SIOCGARP	Gets the arp table entries. The *data* parameter is a pointer to an arpreq structure, as defined in <NET\IF_ARP.H>. The arp table entry passed as an argument is returned from the arp tables if it exists.
SIOCGARP_TR	Gets the token-ring arp table entries with routing information field. The *data* parameter is a pointer to an arpreq_tr structure, as defined in <NET\IF_ARP.H>. The arp table entry from the arp table is returned if it exists.
SIOCGIFADDR	Gets the network interface address. The *data* parameter is a pointer to an ifreq structure, as defined in <NET\IF.H>. The interface address is returned in the argument.
SIOCGIFBOUND	Checks the number of Medium Access Control (MAC) drivers that will be bound or have been bound. The *data* parameter is a pointer to a bndreq structure, defined in <SYS\IOCTLOS2.H>. The *bindinds* variable in the structure will return the number of MAC drivers that the INET protocol will bind to, and the *bound* variable will return the number of MAC drivers that have been bound.
SIOCGIFBRDADDR	Gets the network interface broadcast address. The *data* parameter is a pointer to an ifreq structure, as defined in <NET\IF.H>. The interface broadcast address is returned in the argument.
SIOCGIFCONF	Gets the network interface configuration. The *data* parameter is a pointer to an ifconf structure, as defined in <NET\IF.H>. The interface configuration is returned in the argument. It is important to note that the ifconf structure changed in TCP/IP 4.21.

SIOCGIFDSTADDR	Gets the network interface destination address. The *data* parameter is a pointer to an ifreq structure, as defined in <NET\IF.H>. The interface destination (point-to-point) address is returned in the argument.
SIOCGIFEFLAGS	Gets extended flags for the interface. The *data* parameter is a pointer to an ifreq structure, as defined in <NET\IF.H>. The interface extended flags are returned in the ifr_eflags field.
SIOCGIFFLAGS	Gets the network interface flags. The *data* parameter is a pointer to an ifreq structure, as defined in <NET\IF.H>. The interface flags are returned in the ifr_flags field.
SIOCGIFMETRIC	Gets the network interface routing metric. The *data* parameter is a pointer to an ifreq structure, as defined in <NET\IF.H>. The interface routing metric is returned in the ifr_metric field.
SIOCGIFMTU	Gets the interface MTU value. The *data* parameter is a pointer to an ifreq structure, as defined in <NET\IF.H>. The interface MTU is returned in the ifr_metric field.
SIOCGIFNETMASK	Gets the network interface network mask. The *data* parameter is a pointer to an ifreq structure, as defined in <NET\IF.H>. The interface network mask is returned in the ifr_dstaddr field.
SIOCGIFTRACE	Gets data from the interface input/output tracing buffer. The *data* parameter is a pointer to an ifreq structure, as defined in <NET\IF.H>. The *ifr_data* field should point to the pkt_trace_hdr structure as defined in <NET\IF.H>.
SIOCGIFVALID	Checks if the interface is valid. The *data* parameter is a pointer to an ifreq structure, as defined in <NET\IF.H>.
SIOCGMCAST	Gets the joined multicast addresses for the interface. The *data* parameter is a pointer to an addrreq structure, as defined in <NET\IF.H>.
SIOCGMSL	Gets the TCP Maximum Segment Lifetime (MSL) value, in seconds.
SIOCGSTAT	Gets the serial link interface statistics. The *data* parameter is a pointer to an ifreq structure, as defined in <NET\IF.H>. The ifr_data field should point to an ifstat structure, as defined in <NET\IF.H>.
SIOCGUNIT	Gets the interface unit number.
SIOCMULTISBC	Use broadcast for physical transmission of IP multicast datagrams (RFC 1469).
SIOCMULTISFA	Use functional address for physical transmission of IP multicast datagrams (RFC 1469).
SIOCSARP	Sets an arp table entry. The *data* parameter is a pointer to an *arpreq* as defined in <NET\IF_ARP.H>. The arp table entry passed as an argument is added to the arp tables.
SIOCSARP_TR	Sets a token-ring arp table entry with routing information. The *data* parameter is a pointer to an arp_req structure, as defined in <NET\IF_ARP.H>.
SIOCSHOSTID	Sets the IP address of the host that will be displayed by the HOSTID.EXE utility. The *data* parameter is a pointer to the IP address of the type of unsigned long.
SIOCSIF802_3	Sets the interface to send packets in 802.3 format. The *data* parameter is a pointer to an ifreq structure, as defined in <NET\IF.H>.
SIOCSIFADDR	Sets the network interface address. The *data* parameter is a pointer to an ifreq structure, as defined in <NET\IF.H>. Sets the interface address to the value passed in the argument.
SIOCSIFALLRTB	Sets the interface to use all-route broadcast, for token ring only. The *data* parameter is a pointer to an ifreq structure, as defined in <NET\IF.H>.
SIOCSIFBRD	Sets the interface to use single route broadcast, for token ring only. The *data* parameter is a pointer to an ifreq structure, as defined in <NET\IF.H>.
SIOCSIFBRDADDR	Sets the network interface broadcast address. The *data* parameter is a pointer to an ifreq structure, as defined in <NET\IF.H>. Sets the interface broadcast address to the value passed in the argument.
SIOCSIFDSTADDR	Sets the network interface destination address. The *data* parameter is a pointer to an ifreq structure, as defined in <NET\IF.H>. Sets the interface destination (point-to-point) address to the value passed in the argument.
SIOCSIFEFLAGS	Sets extended flags for the interface. The *data* parameter is a pointer to an ifreq structure, as defined in <NET\IF.H>. The extended flags should be passed in the ifr_eflags field.

SIOCSIFFDDI	Sets the token-ring interface to use canonical format of ARP. The *data* parameter is a pointer to an ifreq structure, as defined in <NET\IF.H>.
SIOCSIFFLAGS	Sets the network interface flags. The *data* parameter is a pointer to an ifreq structure, as defined in <NET\IF.H>. Sets the interface flags to the values passed in the ifr_flags field.
SIOCSIFMETRIC	Sets the network interface routing metric. The *data* parameter is a pointer to an ifreq structure, as defined in <NET\IF.H>. Sets the interface routing metric to the value passed in the ifr_metric field.
SIOCSIFMTU	Sets the interface MTU value. The *data* parameter is a pointer to an ifreq structure, as defined in <NET\IF.H>. Sets the interface MTU to the value passed in the ifr_metric field.
SIOCSIFNETMASK	Sets the network interface network mask. The *data* parameter is a pointer to an ifreq structure, as defined in <NET\IF.H>. Sets the interface network mask to the value passed in the argument.
SIOCSIFNO802_3	Sets the interface to send packets with Ethernet header frame format. The *data* parameter is a pointer to an ifreq structure, as defined in <NET\IF.H>.
SIOCSIFNOFDDI	Sets the token-ring interface to use noncanonical format for ARP.
SIOCSIFNOREDIR	Disable ICMP redirect function for an interface. The *data* parameter is a pointer to an ifreq structure, as defined in <NET\IF.H>.
SIOCSIFRUNNINGBLK	(Token ring only.) Blocks the calling thread until the interface is back in running state. The *data* parameter is a pointer to an ifreq structure, as defined in <NET\IF.H>. This is typically useful when the network cable needs to be disconnected temporarily.
SIOCSIFTRACE	Creates an interface input/output tracing packet. The *data* parameter is a pointer to an ifreq structure, as defined in <NET\IF.H>. The trace packet should be placed in the pkt_trace_hdr structure, as defined in <NET\IF.H>, which should be pointed to by the ifr_data field.
SIOCSIFYESREDIR	Enables ICMP redirect function for an interface. The *data* parameter is a pointer to an ifreq structure, as defined in <NET\IF.H>.
SIOCSMSL	Sets the TCP Maximum Segment Lifetime (MSL) value in seconds.
SIOCSRDBRD	Enables loopback for broadcast packets.
SIOCSSTAT	Sets the serial link interface statistics. The *data* parameter is a pointer to an ifreq structure, as defined in <NET\IF.H>. The statistics are returned in an ifstat structure pointed to by the ifr_data field.
SIOCSSYN	Sets the SYN attack prevention function flag on or off. The *data* parameter should point to an integer that contains zero for off and nonzero for on. The function is off by default.
SIOFLUSHRT	Flushes the entire routing table, including all routes to all interfaces.
SIOFLUSHRTIFP	Flushes all routes for the specified interface only. The *data* parameter is a pointer to an interface name, such as lan0.
SIOSTATCNTAT	Gets the count of ARP entries.
SIOSTATCNTRT	Gets the count of entries in the routing table.
SIOSTATICMP	Gets ICMP statistics. The *data* parameter is a pointer to an icmpstat structure, as defined in <NETINET\ICMP_VAR.H>.
SIOSTATICMPZ	Clears ICMP statistics. The *data* parameter is a pointer to an icmpstat structure, as defined in <NETINET\ICMP_VAR.H>.
SIOSTATIGMP	Gets IGMP statistics. The *data* parameter is a pointer to an igmpstat structure, as defined in <NETINET\IGMP_VAR.H>.
SIOSTATIGMPZ	Clears IGMP statistics. The *data* parameter is a pointer to an igmpstat structure, as defined in <NETINET\IGMP_VAR.H>.
SIOSTATIP	Gets IP statistics. The *data* parameter is a pointer to an ipstat structure, as defined in <NETINET\IP_VAR.H>.
SIOSTATIPZ	Clears IP statistics. The *data* parameter is a pointer to an ipstat structure, as defined in <NETINET\IP_VAR.H>.
SIOSTATMBUF	Gets memory usage status. The *data* parameter is a pointer to an mbstat structure, as defined in

	<SYS\MBUF.H>.
SIOSTATTCP	Gets TCP statistics. The *data* parameter is a pointer to a tcpstat structure, as defined in <NETINET\TCP_VAR.H>.
SIOSTATTCPZ	Clears TCP statistics. The *data* parameter is a pointer to a tcpstat structure, as defined in <NETINET\TCP_VAR.H>.
SIOSTATUDP	Gets UDP statistics. The *data* parameter is a pointer to a udpstat structure, as defined in <NETINET\UDP_VAR.H>.
SIOSTATUDPZ	Clears UDP statistics. The *data* parameter is a pointer to a udpstat structure, as defined in <NETINET\UDP_VAR.H>.

NetBIOS: The following ioctl() calls are supported for the NetBIOS domain:

Option	**Description**
SIOCGNBNAME	Gets the NetBIOS host name. The *data* parameter is a pointer to a socaddr_nb structure, which is defined in <NETNB\NB.H>.
SIOCGNCBFN	Issues ncb.find.name. The *data* parameter is a pointer to a socaddr_nb structure, which is defined in <NETNB\NB.H>.
SIOCSNBNAME	Sets the NetBIOS host name. The *data* parameter is a pointer to a socaddr_nb structure, which is defined in <NETNB\NB.H>.

Return Values

The value 0 indicates success; the value -1 indicates an error. You can get the specific error code by calling sock_errno() or psock_errno().

sock_errno() Value	**Description**
SOCENOTSOCK	The *s* parameter is not a valid socket descriptor.
SOCEINVAL	The request is not valid or not supported.
SOCEOPNOTSUPP	The operation is not supported on the socket.
SOCEFAULT	Using *data* would result in an attempt to access memory outside the caller address space.

Examples

The following is an example of the ioctl() call.

```
int s;
int dontblock;
int rc;
int ioctl(int s, int cmd, caddr_t data);  /* extracted from sys\socket.h */
...
/* Place the socket into nonblocking mode */
dontblock = 1;
rc = ioctl(s, FIONBIO, (char *) &dontblock);
...
```

Related Calls

os2_ioctl()
sock_errno()

----

listen()

The listen() socket call completes the binding necessary for a socket to accept connections and creates a connection request queue for incoming requests.

Syntax

```
#include <types.h>
```

```
#include <sys\socket.h>
#include <netinet\in.h>
int listen(s, backlog)
int s;
int backlog;
```

Parameters

s
Socket descriptor.

backlog
Controls the maximum queue length for pending connections.

Description

The listen() call performs two tasks:

1. Completes the binding necessary for a socket *s*, if bind() has not been called for *s*

2. Creates a connection request queue of length *backlog,* to queue incoming connection requests.
When the queue is full, additional connection requests are ignored.

The listen() call indicates a readiness to accept client connection requests. It transforms an active socket into a passive socket. After listen() is called, *s* can never be used as an active socket to initiate connection requests. listen() is called after allocating a socket with socket() and after binding a name to *s* with bind(). listen() must be called before calling accept().

listen() can only be called on connection-oriented sockets.

If the *backlog* parameter is less than 0, then listen() interprets *backlog* as 0. If the *backlog* parameter is greater than SOMAXCONN, as defined in <SYS\SOCKET.H>, then listen() interprets *backlog* as SOMAXCONN.

Return Values

The value 0 indicates success, the value -1 indicates an error. You can get the specific error code by calling sock_errno() or psock_errno().

Error Code	**Description**
SOCENOTSOCK	The *s* parameter is not a valid socket descriptor.
SOCEOPNOTSUPP	The *s* parameter is not a socket descriptor that supports the listen() call.

Related Calls

accept()
accept_and_recv()
bind()
connect()
sock_errno()
socket()

----

os2_ioctl()

The os2_ioctl() socket call performs special operations on sockets; in particular, operations related to returning status from kernel.

Syntax

```
#include <types.h>
#include <sys\socket.h>
#include <sys\ioctl.h>
#include <net\route.h>
#include <net\if.h>
#include <net\if_arp.h>
int os2_ioctl(s, cmd, data, lendata)
int s;
int cmd;
caddr_t data;
```

```
int lendata;
```

Parameters

s
Socket descriptor.

cmd
Command to perform.

data
Pointer to the data buffer associated with *cmd* where returned data is placed.

lendata
Length (in bytes) of the data to be returned in the buffer.

Description

The following os2_ioctl() commands are supported for the internet domain. The *data* parameter is a pointer to data associated with the particular command, and its format depends on the command that is requested.

Option	Description
SIOSTATARP	Gets the ARP table. The *data* parameter is a pointer to an oarptab structure as defined in <NETINET\IF_ETHER.H>.
SIOSTATAT	Gets all interface addresses. The *data* parameter is a pointer to the buffer for receiving returned data. At return, the first two bytes of the buffer contain the number of returned addresses, followed by the address information for each interface address. For each address, the buffer contains: • The IP address, of type unsigned long • An interface index, of type unsigned short • A netmask, of type unsigned long • The broadcast address, of type unsigned long
SIOSTATIF	Gets interface statistics. The *data* parameter is a pointer to an ifmib structure as defined in <NET\IF.H>.
SIOSTATIF42	Gets interface statistics for all interfaces (maximum of 42). The *data* parameter is a pointer to sequential instances of an ifmib structure as defined in <NET\IF.H>.
SIOSTATRT	Gets routing entries from the routing table. The *data* parameter is a pointer to an rtentries structure as defined in <NET\ROUTE.H>.
SIOSTATSO	Gets sockets' statistics. The *data* parameter is a pointer to sequential instances of a sostats structure as defined in <SYS\SOCKET.H>.

Return Values

The value 0 indicates success; the value -1 indicates an error. You can get the specific error code by calling sock_errno() or psock_errno().

sock_errno() Value	Description
SOCENOTSOCK	The *s* parameter is not a valid socket descriptor.
SOCEINVAL	The request is not valid or not supported.
SOCEOPNOTSUPP	The operation is not supported on the socket.
SOCEFAULT	Using *data* and *lendata* would result in an attempt to access memory outside the caller address space.

Examples

The following is an example of the os2_ioctl() call.

```
int s;
char buf [1024];
int rc;
int os2_ioctl(int s, int cmd, caddr_t data, int lendata);  /* extracted from sys\socket.h */
...
rc = os2_ioctl(s, SIOSTATAT, (char *) buf, sizeof(buf));
...
```

Related Calls

sock_errno()
ioctl()

----

os2_select()

The socket call gets read, write, and exception status on a group of sockets.

With the os2_select() call, the socket numbers are specified as an array of integers, in which the read socket numbers are followed by write socket numbers, followed by the exception socket numbers. TCP/IP for OS/2 Warp monitors the activity on a socket by specifying the number of sockets to be checked for readability, readiness for writing, and exception-pending conditions.

Syntax

```
#include <types.h>
#include <unistd.h>
int os2_select(s, noreads, nowrites, noexcepts, timeout)
int *s;
int noreads;
int nowrites;
int noexcepts;
long timeout;
```

Parameters

s
Pointer to an array of socket numbers where the read socket numbers are followed by the write socket numbers, and then followed by the exception socket numbers.

noreads
Number of sockets to be checked for readability.

nowrites
Number of sockets to be checked for readiness for writing.

noexcepts
Number of sockets to be checked for exception-pending conditions. The only exception-pending condition is out-of-band data in the receive buffer.

timeout
Maximum interval, in milliseconds, to wait for the selection to complete.

Description

This call monitors activity on a set of different sockets until a timeout expires, to see if any sockets are ready for reading or writing, or if any exceptional conditions are pending.

If the timeout value is 0, select() does not wait before returning. If the timeout value is -1, select() does not time out, but returns when a socket becomes ready. If the timeout value is a number of milliseconds, select() waits for the specified interval before returning. The select() call checks all indicated sockets at the same time and returns when any of them is ready.

Reinitializing the socket array every time select() is called is required.

Return Values

The number of ready sockets is returned. The value -1 indicates an error. The value 0 indicates an expired time limit. If the return value is greater than 0, the socket numbers in *s* that were not ready are set to -1. You can get the specific error code by calling sock_errno() or psock_errno().

Error Code	**Description**
SOCENOTSOCK	The *s* parameter is not a valid socket descriptor.
SOCEFAULT	The address is not valid.
SOCEINVAL	Invalid argument.
SOCEINTR	Interrupted system call.

Examples

The following is an example of the os2_select() call.

```
#define MAX_TIMEOUT  1000
/* input_ready(insock)- Check to see if there is available input on
 * socket insock.
 * Returns 1 if input is available.
 *         0 if input is not available.
 *        -1 on error.
 */

int input_ready(insock)
int insock;                 /* input socket descriptor */

{
  int socks[1];    /* array of sockets */
  long timeout = MAX_TIMEOUT;

  /* put socket to check in socks[] */
  socks[0] = insock;

  /* check for READ availability on this socket */
  return os2_select(socks, 1, 0, 0, timeout);
}
```

Related Calls

accept()
accept_and_recv()
connect()
recv()
select()
send()
sock_errno()
socket()

----

psock_errno()

The psock_errno() socket call writes a short error message to the standard error device.

Syntax

```
#include <sys/socket.h>
void psock_errno(s)
char *s;
```

Parameters

s
Pointer to a buffer.

Description

This call writes a short error message to the standard error display describing the last error encountered during a call to a socket library function. If *s* is not a NULL pointer and does not point to a null string, the string it points to is printed, followed by a colon, followed by a space, followed by the message. If *s* is a NULL pointer or points to a null string, only the message is printed.

The error code is acquired by calling sock_errno(). The error code is set when errors occur. Subsequent socket calls do not clear the error code.

Related Calls

sock_errno()

----

readv()

The readv() socket call receives data on a socket into a set of buffers.

Syntax

```
#include <types.h>
#include <sys/uio.h>
int readv(s, iov, iovcnt)
int s;
struct iovec *iov;
int iovcnt;
```

Parameters

s
: Socket descriptor.

iov
: Pointer to an array of iovec structures.

iovcnt
: Number of iovec structures pointed to by the *iov* parameter. The maximum number of iovec structures is 1024.

Description

This call reads data on a socket with descriptor *s* and stores it in a set of buffers. The data is scattered into the buffers specified by *iov*[0]...*iov*[*iovcnt*-1]. The iovec structure is defined in <SYS/UIO.H> and contains the following fields:

Field	**Description**
iov_base	Points to the buffer
iov_len	Length of the buffer

The readv() call applies only to connected sockets. For information on how to use readv() with datagram and raw sockets, see Datagram or Raw Sockets.

TCP/IP alters *iov_base* and *iov_len* for each element in the input struct iovec array. *iov_base* will point to the next character of the processed (sent or received) data on the original buffer, and *iov_len* will become (input value - processed length). Thus if only partial data has been sent or received and the application expects more data to send or receive, it can pass the same iovec structure back in a subsequent call.

This call returns up to the number of bytes in the buffers pointed to by the *iov* parameter. This number is the sum of all *iov_len* fields. If less than the number of bytes requested is available, the call returns the number currently available. If data is not available at the socket with descriptor *s*, the readv() call waits for data to arrive and blocks the caller, unless the socket is in nonblocking mode. See ioctl() for a description of how to set nonblocking mode. The UDP sockets can send and receive datagrams as large as 32739 bytes (32 * 1024 -1 - IP header (20 bytes) - UDP header (8 bytes)).

Return Values

When successful, the number of bytes of data received into the buffer is returned. The value -1 indicates an error. You can get the specific error code by calling sock_errno() or psock_errno().

sock_errno() Value	**Description**
SOCENOTSOCK	The *s* parameter is not a valid socket descriptor.
SOCEFAULT	Using *iov* and *iovcnt* would result in an attempt to access memory outside the caller's address space.
SOCEINTR	Interrupted system call.
SOCEINVAL	*iovcnt* was not valid, or one of the fields in the *iov* array was not valid.
SOCEWOULDBLOCK	The *s* parameter is in nonblocking mode and no data is available to read, *or* the

SO_RCVTIMEO option has been set for socket *s* and the timeout expired before any data arrived to read.

Related Calls

accept()
accept_and_recv()
connect()
getsockopt()
ioctl()
recv()
recvfrom()
recvmsg()
select()
send()
sendto()
setsockopt()
so_cancel()
sock_errno()
socket()
writev()

----

recv()

The socket call receives data on a connected socket.

Syntax

```
#include <types.h>
#include <sys\socket.h>
int recv(s, buf, len, flags)
int s;
char *buf;
int len;
int flags;
```

Parameters

s
: Socket descriptor.

buf
: Pointer to the buffer that receives the data.

len
: Length of the buffer in bytes pointed to by the *buf* parameter.

flags
: Permits the call to exercise control over the reception of messages. Set this parameter by specifying one or more of the following flags. If you specify more than one flag, use the logical OR operator (|) to separate them. Setting this parameter is supported only for sockets in the internet domain.

MSG_DONTWAIT	Do not wait for resources or data during this call.
MSG_OOB	Reads any out-of-band data on the socket.
MSG_PEEK	Peeks at the data present on the socket; the data is returned but not consumed, so that a subsequent receive operation sees the same data.
MSG_WAITALL	Wait for data to fill all buffers before returning.

Description

This call receives data on a socket with descriptor *s* and stores it in the buffer pointed to by *buf*. The recv() call applies only to connected sockets. For information on how to use recv() with datagram and raw sockets, see Datagram or Raw Sockets.

The recv() call returns the length of the incoming data. If a datagram or sequenced packet is too long to fit in the buffer, the excess is discarded. No data is discarded for stream or sequenced packet sockets. If data is not available at the socket with descriptor *s*, the recv() call waits for data to arrive and blocks the caller, unless the socket is in nonblocking mode. See ioctl() for a description of how to set nonblocking mode.

Use the select() call to determine when more data arrives.

Return Values

When successful, the number of bytes of data received into the buffer is returned. The value 0 indicates that the connection is closed. The value -1 indicates an error. You can get the specific error code by calling sock_errno() or psock_errno().

Error Code	**Description**
SOCENOTSOCK	The *s* parameter is not a valid socket descriptor.
SOCEFAULT	Using the *buf* and *len* parameters would result in an attempt to access memory outside the caller's address space.
SOCEINTR	Interrupted system call.
SOCEINVAL	Invalid argument.
SOCEWOULDBLOCK	The *s* parameter is in nonblocking mode and no data is available to receive, or the SO_RCVTIMEO option has been set for socket *s* and the timeout expired before any data arrived to receive.

Related Calls

connect()
getsockopt()
ioctl()
readv()
recvfrom()
recvmsg()
select()
send()
sendmsg()
sendto()
setsockopt()
shutdown()
sock_errno()
socket()
writev()

----

recvfrom()

The socket call receives data on a socket.

Syntax

```
#include <types.h>
#include <sys\socket.h>
int recvfrom(s, buf, len, flags, name, namelen)
int s;
char *buf;
int len;
int flags;
struct sockaddr *name;
int *namelen;
```

Parameters

s

Socket descriptor.

buf
Pointer to the buffer that receives the data.

len
Length of the buffer in bytes pointed to by the *buf* parameter.

flags
Permits the call to exercise control over the reception of messages. Set this parameter by specifying one or more of the following flags. If you specify more than one flag, use the logical OR operator (|) to separate them. Setting this parameter is supported only for sockets in the internet domain.

MSG_DONTWAIT	Do not wait for resources or data during this call.
MSG_OOB	Reads any out-of-band data on the socket.
MSG_PEEK	Peeks at the data present on the socket; the data is returned but not consumed, so that a subsequent receive operation sees the same data.
MSG_WAITALL	Wait for data to fill all buffers before returning.

name
Pointer to a sockaddr structure (buffer) that data is received from. If *name* is a nonzero value, the source address is returned.

namelen
Pointer to the size in bytes of the buffer pointed to by *name* .

Description

The recvfrom() call receives data on a socket with descriptor *s* and stores it in a buffer. The recvfrom() call applies to any socket type, whether connected or not.

If *name* is nonzero, the address of the data sender is returned. The *namelen* parameter is first initialized to the size of the buffer associated with *name* ; on return, it is modified to indicate the actual number of bytes stored there.

The recvfrom() call returns the length of the incoming message or data. If a datagram or sequenced packet is too long to fit in the supplied buffer, the excess is discarded. No data is discarded for stream or sequenced packet sockets. If data is not available at the socket with descriptor *s* , the recvfrom() call waits for data to arrive and blocks the caller, unless the socket is in nonblocking mode. See ioctl() for a description of how to set nonblocking mode.

Return Values

When successful, the number of bytes of data received into the buffer is returned. The value -1 indicates an error. You can get the specific error code by calling sock_errno() or psock_errno().

Error Code	**Description**
SOCENOTSOCK	The *s* parameter is not a valid socket descriptor.
SOCEFAULT	Using the *buf* and *len* parameters would result in an attempt to access memory outside the caller's address space.
SOCEWOULDBLOCK	The *s* parameter is in nonblocking mode and no data is available to receive, or the SO_RCVTIMEO option has been set for socket *s* and the timeout expired before any data arrived to receive.
SOCEINVAL	Invalid argument.

Related Calls

ioctl()
readv()
recv()
recvmsg()
select()
send()
sendmsg()
sendto()
setsockopt()
shutdown()
sock_errno()
socket()

writev()

----

recvmsg()

The socket call receives data and control information on a specified socket.

Syntax

```
#include <types.h>
#include <sys\socket.h>
int recvmsg(s, msg, flags)
int s;
struct msghdr *msg;
int flags;
```

Parameters

s
Socket descriptor.

msg
Pointer to a message header that receives the message.

flags
Permits the call to exercise control over the reception of messages. Set this parameter by specifying one or more of the following flags. If you specify more than one flag, use the logical OR operator (|) to separate them. Setting this parameter is supported only for sockets in the internet domain.

MSG_DONTWAIT	Do not wait for resources or data during this call.
MSG_OOB	Reads any out-of-band data on the socket.
MSG_PEEK	Peeks at the data present on the socket; the data is returned but not consumed, so that a subsequent receive operation will see the same data.
MSG_WAITALL	Wait for data to fill all buffers before returning.

Description

This call receives a message on a socket with descriptor *s*.

Networking services supports the following msghdr structure.

Note: The fields *msg_control* and *msg_controllen* are ignored for the NetBIOS and Local IPC domains.

```
struct msghdr {
       caddr_t msg_name;             /* optional pointer to destination address buffer */
       int     msg_namelen;          /* size of address buffer */
       struct  iovec  *msg_iov;      /* scatter/gather array */
       int     msg_iovlen;           /* number of elements in msg_iov, maximum 1024 */
       caddr_t msg_control;          /* ancillary data */
       u_int   msg_controllen;       /* ancillary data length */
       int     msg_flags;            /* flags on receive message */
};
```

msg_iov is a scatter/gather array of iovec structures. The iovec structure is defined in <SYS/UIO.H> and contains the following fields:

Field	**Description**
iov_base	Pointer to the buffer
iov_len	Length of the buffer

TCP/IP alters *iov_base* and *iov_len* for each element in the input struct iovec array. *iov_base* will point to the next character of the processed (sent or received) data on the original buffer, and *iov_len* will become (input value - processed length). Thus if only partial data has been sent or received and the application expects more data to send or receive, it can pass the same iovec structure back in a subsequent call.

The recvmsg() call applies to connection-oriented or connectionless sockets.

This call returns the length of the data received. If a datagram or sequenced packet is too long to fit in the supplied buffer, the excess is discarded. No data is discarded for stream sockets. If data is not available at the socket with descriptor *s*, the recvmsg() call waits for a message to arrive and blocks the caller, unless the socket is in nonblocking mode. See ioctl() for a description of how to set nonblocking mode.

Return Values

When successful, the number of bytes of data received into the buffer is returned. The value 0 indicates the connection is closed; the value -1 indicates an error. You can get the specific error code by calling sock_errno() or psock_errno().

Error Code	**Description**
SOCENOTSOCK	The *s* parameter is not a valid socket descriptor.
SOCEFAULT	Using *msg* would result in an attempt to access memory outside the caller's address space.
SOCEINTR	Interrupted system call.
SOCENOTCONN	The socket is not connected.
SOCEWOULDBLOCK	The *s* parameter is in nonblocking mode, and no data is available to receive.

Examples

The following is an example of using recvmsg() call to receive token ring routing information in the msg_control buffers:

Example of recvmsg() Call

```
char buf[50], control_buf[100];
main(int argc, char *argv[])
{
    struct sockaddr_in  server;
    int optlen, smsg, byterecv,rv,i, ip_recvtrri;
    struct msghdr msg;
    struct cmsghdr *cmsg;
    struct iovec iov;
    struct timeval tv;

    if ((smsg = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP)) < 0)
        psock_errno("Socket()");

    server.sin_len    = sizeof(struct sockaddr);
    server.sin_family = AF_INET;
    server.sin_port   = htons(atoi(argv[1]));         /* port number */
    server.sin_addr.s_addr = INADDR_ANY;

    if (bind(smsg, (struct sockaddr *)&server , sizeof(server)) < 0)
       psock_errno("bind()");

    iov.iov_base = buf;
    iov.iov_len  = sizeof(buf)-1;

    msg.msg_name       = NULL;
    msg.msg_namelen    = sizeof(struct sockaddr);
    msg.msg_iov        = &iov
    msg.msg_iovlen     = 1;
    msg.msg_control    = control_buf;
    msg.msg_controllen = sizeof(struct cmsghdr)+4+18;/*4 byte ipaddr + 18 TRRI*/

    ip_recvtrri  = 1;
    if (setsockopt(smsg,IPPROTO_IP,IP_RECVTRRI,(char *)&ip_recvtrri ,
              sizeof(ip_recvtrri)) < 0)
        psock_errno("setsockopt() IP_RECVTRRI");

    /* Set another IP socket options for timeout so we do not block waiting */
    tv.tv_sec  = 10;  /* Wait for max 10 sec on recvmsg */
    tv.tv_usec = 0;
    rv = setsockopt(smsg, SOL_SOCKET, SO_RCVTIMEO, (char *) &tv,
             sizeof(struct timeval));
    if (rv < 0) psock_errno("Set SO_RCVTIMEO");

    if((byterecv=recvmsg(smsg, &msg, 0))<0)
       psock_errno("recvmsg()");
    else {
      cmsg = (struct cmsghdr *) msg.msg_control;

      printf(" IP_RECV TR RI (data in network byte order): ");
      for (i=sizeof(struct cmsghdr); i < cmsg->cmsg_len;i++)
         printf(" %x",msg.msg_control[i]);
    }

    soclose(smsg);
}
```

Related Calls

- connect()
- getsockopt()
- ioctl()
- os2_ioctl()
- recv()
- recvfrom()
- select()
- send()
- sendmsg()
- sendto()
- shutdown()
- sock_errno()
- socket()

----

removesocketfromlist()

The removesocketfromlist() call removes a socket from the list of owned sockets for the calling process.

Syntax

```
#include <types.h>
#include <sys\socket.h>
int removesockettolist(s)
int s;
```

Parameters

s
Socket descriptor.

Description

When a process ends, the sockets library automatically cleans up sockets by registering an exit list handler. This exit routine closes all open sockets that are maintained in a process's socket list. When a process is initiated the list is empty, and whenever a socket() or soclose() call is made the list is updated. The removesocketfromlist() call provides a mechanism to transfer socket ownership to another process: it removes the socket indicated by the *s* parameter from the calling process's socket ownership list.

Return Values

The value 1 indicates success; the value 0 indicates that the socket could not be found in the list.

Related Calls

- addsockettolist()

----

select()

The socket call gets read, write, and exception status on a group of sockets.

The BSD version monitors the activity on sockets by specifying an array (fd_set) of socket numbers for which the caller wants to read the data, write the data, and check exception-pending conditions. The BSD version provides FD_SET, FD_CLR, FD_ISSET, and FD_ZERO macros to add or delete socket numbers from the array.

Syntax

```
#include <types.h>
#include <unistd.h>
#include <sys\time.h>
int select(nfds, readfds, writefds, exceptfds, timeout)
int nfds;
fd_set *readfds;
fd_set *writefds;
fd_set *exceptfds;
struct timeval *timeout;
```

Parameters

nfds
: This parameter is unused; it is maintained for compatibility with BSD.

readfds
: Pointer to a list of descriptors to be checked for reading.

writefds
: Pointer to a list of descriptors to be checked for writing.

exceptfds
: Pointer to a list of descriptors to be checked for exception-pending conditions. For networking services sockets, the only exception-pending condition is out-of-band data in the receive buffer.

timeout
: Pointer to the time to wait for the select() call to complete.

Description

This call monitors activity on a set of different sockets until a timeout expires, to see if any sockets are ready for reading or writing, or if any exception-pending conditions are pending.

Reinitializing *readfds* , *writefds* , and *exceptfds* every time select() is called is required.

If timeout is a NULL pointer, the call blocks indefinitely until one of the requested conditions is satisfied. If timeout is non-NULL, it specifies the maximum time to wait for the call to complete. To poll a set of sockets, the timeout pointer should point to a zeroed timeval structure. The timeval structure is defined in the <SYS\TIME.H> header file and contains the following fields:

```
struct timeval {
       long tv_sec;  /* Number of seconds */
       long tv_usec; /* Number of microseconds */
}
```

An fd_set is made up of an array of integers. Macros are provided to manipulate the array.

Macro	Description
FD_SET(*socket, array_address*)	Adds the socket to the list pointed to by *array_address* .
FD_CLR(*socket, array_address*)	Removes the socket from the list.
FD_ISSET(*socket, array_address*)	Returns true if the descriptor is part of the array; otherwise, returns false.

FD_ZERO(*socket, array_address*)	Clears the entire array for all socket descriptors.

Note: For macros FD_SET, FD_CLR, FD_ISSET, and FD_ZERO, define the parameters *socket* and *array_address* in the following manner:

```
    int socket;
    struct fd_set *array_address;
```

Setting any of the descriptor pointers to zero indicates that no checks are to be made for the conditions. For example, setting *exceptfds* to be a NULL pointer causes the select call to check for only read and write conditions.

Return Values

The total number of ready sockets (in all arrays) is returned. The value -1 indicates an error. The value 0 indicates an expired time limit. If the return value is greater than 0, the socket descriptors in each array that are not ready are removed from the array and fd_array is rearranged so that the ready sockets are at the top. The *fd_count* parameter is adjusted accordingly and returned. You can get the specific error code by calling sock_errno() or psock_errno().

sock_errno() Value	Description
SOCENOTSOCK	The *s* parameter is not a valid socket descriptor.
SOCEFAULT	The address is not valid.
SOCEINVAL	Invalid argument.

Examples

Following is an example of the select() call.

```
...
fd_set readsocks;
fd_set writesocks;
fd_set exceptsocks;
struct timeval timeout;
int number_found;
...
/* add socket to read/write/except arrays.  To add descriptor s use
 *   FD_SET (s, &readsocks);
 *
 */
...
number_found = select(0,&readsocks, &writesocks,
                      &exceptsocks, &timeout);
```

Related Calls

- accept()
- accept_and_recv()
- connect()
- os2_select()
- recv()
- send()
- sock_errno()
- socket()

----

send()

The socket call sends data on a connected socket.

Syntax

```
#include <types.h>
#include <sys\socket.h>
```

```
int send(s, msg, len, flags)
int s;
char *msg;
int len;
int flags;
```

Parameters

s
: Socket descriptor.

msg
: Pointer to a buffer containing the message to transmit.

len
: Length of the message pointed to by the *msg* parameter.

flags
: Allows the sender to control the transmission of the message. Set this parameter by specifying one or more of the following flags. If you specify more than one flag, use the logical OR operator (|) to separate them. Setting this parameter is supported only for sockets in the internet domain.

MSG_DONTROUTE	The SO_DONTROUTE socket option is turned on for the duration of the operation. This is usually used only by diagnostic or routing programs.
MSG_DONTWAIT	Do not wait for resources during this call.
MSG_EOF	Indicates that the sending of data on the connection is complete. This flag is effective on T/TCP connections only.
MSG_OOB	Sends out-of-band data on sockets that support SOCK_STREAM communication.

Description

This call sends data on the socket with descriptor *s*. The send() call applies to connected sockets. For information on how to use send() with datagram and raw sockets, see Datagram or Raw Sockets. The sendto() and sendmsg() calls can be used with unconnected or connected sockets.

To broadcast on a socket, first issue a setsockopt() call using the SO_BROADCAST option to gain broadcast permission.

Specify the length of the message with the *len* parameter. If the message is too long to pass through the underlying protocol, the system returns an error and does not transmit the message.

No indication of failure to deliver is implied in a send() call. A return value of -1 indicates some locally detected errors.

If buffer space is not available at the socket to hold the message to be sent, the send() call normally blocks, unless the socket is placed in nonblocking mode. See ioctl() for a description of how to set nonblocking mode. Use the select() call to determine when it is possible to send more data.

Return Values

When successful, the number of bytes of the socket with descriptor *s* that is added to the send buffer is returned. This may be less than the number of bytes specified in the length parameter. Successful completion does not imply that the data has already been delivered to the receiver. The return value -1 indicates an error was detected on the sending side of the connection. You can get the specific error code by calling sock_errno() or psock_errno().

Error Code	**Description**
SOCENOTSOCK	The *s* parameter is not a valid socket descriptor.
SOCEFAULT	Using the *msg* and *len* parameters would result in an attempt to access memory outside the caller's address space.
SOCEINTR	Interrupted system call.
SOCEINVAL	Invalid argument.
SOCENOBUFS	No buffer space is available to send the message.
SOCEWOULDBLOCK	The *s* parameter is in nonblocking mode and the data cannot be sent without blocking, or the SO_SNDTIMEO option has been set for socket *s* and the timeout expired before any

data was sent.

Related Calls

send_file()
connect()
getsockopt()
ioctl()
readv()
recv()
recvfrom()
recvmsg()
select()
sendmsg()
sendto()
setsockopt()
shutdown()
sock_errno()
socket()
writev()

----

send_file()

The send_file() function sends the file data over a connected socket.

Syntax

```
#include <types.h>
#include <sys\socket.h>
ssize_t send_file(socket_ptr, sf_struct, flags)
int * socket_ptr;
struct sf_parms * sf_struct;
int flags;
```

Parameters

socket_ptr
Pointer to the socket descriptor of a connected socket.

sf_struct
Pointer to a structure that contains variables needed by send_file().

flags
Allows the sender to control the transmission of the message. Set this parameter by specifying one or more of the following flags. If you specify more than one flag, use the logical OR operator (|) to separate them. Setting this parameter is supported only for sockets in the Internet domain.

SF_CLOSE	Close the connection after data has been successfully sent or queued for retransmission.
SF_REUSE	Prepare the socket for reuse after the data has been successfully sent or queued for transmission and the existing connection closed.

Description

The *send_file()* function sends data from the file associated with the open file handle, directly from the file-system cache, over the connection associated with the socket.

The *send_file()* function attempts to write header_length bytes from the buffer pointed to by header_data, followed by file_bytes from the file associated with file_descriptor, followed by trailer_length bytes from the buffer pointed to by trailer_data, over the connection associated with the socket pointed to by socket_ptr.

As data is sent, the kernel updates the variables in the sf_parms structure so that if the send_file() is interrupted by a signal, the application simply needs to reissue the send_file(). If the application sets file_offset > the actual file size, or file_bytes > (the actual file size - file_offset), the return value is -1 with errno set to [EINVAL].

The flags argument is effective only after all the data has been sent successfully; otherwise it is ignored. The application should zero the flags argument before setting the appropriate value. If flags = **SF_REUSE** and socket reuse is not supported, then upon successful completion of sending the data, the kernel closes this socket and sets the socket pointed to by socket_ptr to -1. If flags = **SF_CLOSE** and send_file() completes successfully, the socket pointed to by socket_ptr is set to -1 by the kernel.

Implementation Note

- The **send_file()** API can be used only on OS/2 WARP 4.5, which supports Kernel Execution Environment (KEE32), and Installable File System Mechanism (IFSM32). For more information on KEE, refer to OS/2 WARP 4.5 related Document.
- The performance of **accept_and_recv()** and **send_file()** is greatly effected by the number of threads that are allowed to be active concurrently. If too few threads are active and work is not done quickly enough, too many threads, needless traps, and context switches can reduce performance by a factor of two. The kernel should make some attempt to intelligently determine how many threads may be active concurrently. Optimally, the number of threads active concurrently should be just below the saturation point, that is, one or two connect request waiting on the backlog queue so that when an accept_and_recv() thread completes, there is another connect requests ready to be processed.

Return Values

There are three possible return values from *send_file()* :

- -1: An error has occurred, check errno for more information.
- 0: The command has completed successfully.
- 1: The command was interrupted by a signal while sending data.

Error Code	**Description**
EACCESS	The calling process does not have the appropriate privileges.
EBADF	Either the socket pointed to by the socket_ptr argument, or the file_desc is not a valid descriptor.
ECONNABORTED	A connection has stopped.
ECONNRESET	A connection was forcibly closed by a peer.
EFAULT	The data buffer pointed to by socket_ptr, file_size, header_data or trailer_data was not valid.
EINTR	The send_file() function was interrupted by a signal that was caught before any data was sent.
EINVAL	The value specified by an attribute is not valid.
ENOTCONN	The socket is not connected.
EPIPE	The socket is shutdown for writing, or the socket is connection-mode and no longer connected.
EIO	An I/O error occurred.
ENETDOWN	The local interface used to reach the destination is down.
ENETUNREACH	No route to the destination is present.
ENOBUFS	No buffer space is available.
ENOMEM	There was insufficient memory available to complete the operation.
ENOSR	There were insufficient STREAMS resources available for the operation to complete.
ENOTSOCK	The socket pointed to by the socket_ptr argument does not refer to a socket.

Examples

The following is an example of using **send_file()** call to send a file data over a connected socket.

```
#define MSG_CLOSE 0x800
#define O_RDONLY  0x4

#include #include #include #include #include #include #include #include
char  serveraddress[128],filename[256];
int   serverport = 6000;
int   fd,rc,s;

struct sf_parms
   {
        void   *header_data;      /* ptr to header data */
        size_t header_length;     /* size of header data */
        int    file_handle;       /* file handle to send from */
        size_t file_size;         /* size of file */
        int    file_offset;       /* byte offset in file to send from */
        size_t file_bytes;        /* bytes of file to be sent */
        void   *trailer_data;     /* ptr to trailer data */
        size_t trailer_length;    /* size of trailer data */
        size_t bytes_sent;        /* bytes sent in this send_file call */
   } sfp;

int   putfile (void);

int main (int argc, char *argv[])
{

 strcpy (serveraddress, argv[1]);   /* argv[1] is server address to which file is to be sent */
 strcpy (filename, argv[2]);        /* argv[2] is name of the file to be sent */

 printf ("Sending File to server\n");
   if ((rc = putfile()) != 0)
        {
          printf ("Putfile() failed rc = %d sock_errno = %d \n", rc, sock_errno());
            return(rc);
        }
}

int putfile ()
{
    struct sockaddr_in servername;

    if( (s = socket (PF_INET, SOCK_STREAM, 0)) != -1 )
    {
      servername.sin_len         = sizeof(servername);
      servername.sin_family      = AF_INET;
      servername.sin_addr.s_addr = inet_addr(serveraddress);
      servername.sin_port        = serverport;

    if((rc = connect(s,(struct sockaddr *)&servername,sizeof(servername))) != -1)
       {
            fd =open(filename,O_RDONLY,0);
            sfp.header_data   = 0;
            sfp.header_length = 0;
            sfp.file_handle   = fd;
            sfp.file_size     = -1;
            sfp.file_offset   = 0;
            sfp.file_bytes    = -1;
            sfp.trailer_data  = 0;
            sfp.trailer_length= 0;
            sfp.bytes_sent    = 0;

          if(( rc = send_file(&s,&sfp,MSG_CLOSE)) != 0)
                printf( " ******  FILE NOT SENT  ****** ");
            close(fd);
       }
      else
        printf ("send_file :connect() failed sock_errno = %d \n",sock_errno());
    }
    else
      printf ("send_file :socket() failed rc = %d\n", sock_errno());
   return(rc);
}
```

Related Calls

send()

connect()
getsockopt()
ioctl()
readv()
recv()
recvfrom()
recvmsg()
select()
sendmsg()
sendto()
setsockopt()
shutdown()
sock_errno()
socket()
writev()

----

sendmsg()

The socket call sends data and control information on a specified socket.

Syntax

```
#include <types.h>
#include <sys\socket.h>
int sendmsg(s, msg, flags)
int s;
struct msghdr *msg;
int flags;
```

Parameters

s
Socket descriptor.

msg
Pointer to a message header containing a message to be sent.

flags
Allows the sender to control the message transmission. Set this parameter by specifying one or more of the following flags. If you specify more than one flag, use the logical OR operator (|) to separate them. Setting this parameter is supported only for sockets in the internet domain.

MSG_DONTROUTE	The SO_DONTROUTE socket option is turned on for the duration of the operation. This is usually used only by diagnostic or routing programs.
MSG_DONTWAIT	Do not wait for resources during this call.
MSG_EOF	Indicates that the sending of data on the connection is complete. This flag is effective on T/TCP connections only.
MSG_OOB	Sends out-of-band data on the socket.

Description

This call sends a msghdr structure on a socket with descriptor *s*.

Networking services supports the following msghdr structure.

Note: The fields *msg_control* and *msg_controllen* are ignored for the NetBIOS and Local IPC domains.

```
struct msghdr {
       caddr_t msg_name;             /* optional pointer to destination address buffer */
       int     msg_namelen;          /* size of address buffer */
       struct iovec  *msg_iov;       /* scatter/gather array */
       int     msg_iovlen;           /* number of elements in msg_iov, maximum 1024 */
```

```
          caddr_t msg_control;          /* ancillary data */
          u_int   msg_controllen;       /* ancillary data length */
          int     msg_flags;            /* flags on received message */
   };
```

To broadcast on a socket, the application program must first issue a setsockopt() call using the SO_BROADCAST option, to gain broadcast permission.

The sendmsg() call applies to connection-oriented and connectionless sockets.

msg_iov is a scatter/gather array of iovec structures. The iovec structure is defined in <SYS/UIO.H> and contains the following fields:

Field	**Description**
iov_base	Pointer to the buffer
iov_len	Length of the buffer

TCP/IP alters *iov_base* and *iov_len* for each element in the input struct iovec array. *iov_base* will point to the next character of the processed (sent or received) data on the original buffer, and *iov_len* will become (input value - processed length). Thus if only partial data has been sent or received and the application expects more data to send or receive, it can pass the same iovec structure back in a subsequent call.

This call returns the length of the data sent. If the socket with descriptor *s* is not ready for sending data, the sendmsg() call waits unless the socket is in nonblocking mode. See ioctl() for a description of how to set nonblocking mode.

Return Values

When successful, the number of bytes of data sent is returned. Successful completion does not guarantee delivery of the data to the receiver. The return value -1 indicates an error was detected on the sending side of the connection. You can get the specific error code by calling sock_errno() or psock_errno().

Error Code	**Description**
SOCEDESTADDRREQ	The msghdr *msg_name* parameter is set to NULL and a destination address is required.
SOCENOTSOCK	The *s* parameter is not a valid socket descriptor.
SOCEFAULT	Using *msg* would result in an attempt to access memory outside the caller's address space.
SOCEINTR	Interrupted system call.
SOCEINVAL	*msg_namelen* is not the size of a valid address for the specified address family.
SOCEMSGSIZE	The message was too big to be sent as a single datagram.
SOCENOBUFS	No buffer space is available to send the message.
SOCENOTCONN	The socket is not connected.
SOCEWOULDBLOCK	The *s* parameter is in nonblocking mode and the data cannot be sent without blocking.

Related Calls

getsockopt()
ioctl()
readv()
recv()
recvfrom()
recvmsg()
select()
send()
sendto()
setsockopt()
shutdown()
sock_errno()
socket()
writev()

----

sendto()

The socket call sends data on a socket.

Syntax

```
#include <types.h>
#include <sys\socket.h>
int sendto(s, msg, len, flags, to, tolen)
int s;
char *msg;
int len;
int flags;
struct sockaddr *to;
int tolen;
```

Parameters

s
: Socket descriptor.

msg
: Pointer to the buffer containing the message to transmit.

len
: Length of the message in the buffer pointed to by the *msg* parameter.

flags
: Allows the sender to control the message transmission. Set this parameter to 0, or to one or more of the following flags. If you specify more than one flag, use the logical OR operator (|) to separate them. Setting this parameter is supported only for sockets in the internet domain.

MSG_DONTROUTE	The SO_DONTROUTE socket option is turned on for the duration of the operation. This is usually used only by diagnostic or routing programs.
MSG_DONTWAIT	Do not wait for resources during this call.
MSG_EOF	Indicates that the sending of data on the connection is complete. This flag is effective on T/TCP connections only.
MSG_OOB	Sends out-of-band data on the socket.

to
: Pointer to a sockaddr structure (buffer) containing the destination address.

tolen
: Size in bytes of the buffer pointed to by the *to* parameter.

Description

This call sends data on the socket with descriptor *s*. The sendto() call applies to connected or unconnected sockets. For unconnected datagram and raw sockets, the sendto() call sends data to the specified destination address. For stream and sequenced packet sockets the destination address is ignored.

To broadcast on a socket, first issue a setsockopt() call using the SO_BROADCAST option to gain broadcast permissions.

Provide the address of the target using the *to* parameter. Specify the length of the message with the *tolen* parameter. If the message is too long to pass through the underlying protocol, the error SOCEMSGSIZE is returned and the message is not transmitted.

If the sending socket has no space to hold the message to be transmitted, the sendto() call blocks the message, unless the socket is in a nonblocking I/O mode.

Use the select() call to determine when it is possible to send more data.

Datagram sockets are *connected* by calling connect(). This identifies the peer to send/receive the datagram. Once a datagram socket is connected to a peer, you may still use the sendto() call but a destination address cannot be included.

To change the peer address when using connected datagram sockets, issue a connect() call with a null address. Specifying a null address on a connected datagram socket removes the peer address specification. You can then either issue a sendto() call specifying a different destination address or issue a connect() call to connect to a different peer. For more information on connecting datagram sockets and specifying null addresses, see Datagram or Raw Sockets.

If the *to* parameter is specified and this sendto() call was preceded by a connect() call, the *dst* parameter must be NULL. If not NULL, the error SOCEISCONN is returned and the message is not sent. If the *to* parameter is specified and this sendto() call was not preceded by a connect() call, this sendto() call results in socket *s* being connected to *dst* , the message being sent, and socket *s* being disconnected from *dst* .

Return Values

When successful, the number of bytes of data sent is returned. Successful completion does not guarantee delivery of the data to the receiver. The return value -1 indicates an error was detected on the sending side. You can get the specific error code by calling sock_errno() or psock_errno().

Error Code	**Description**
SOCENOTSOCK	The *s* parameter is not a valid socket descriptor.
SOCEFAULT	Using the *msg* and *len* parameters would result in an attempt to access memory outside the caller's address space.
SOCEINVAL	The *tolen* parameter is not the size of a valid address for the specified address family.
SOCEISCONN	This call was preceded by a connect() call, the *to* parameter of this call is specified, but the *dst* parameter is not NULL.
SOCEMSGSIZE	The message was too big to be sent as a single datagram.
SOCENOBUFS	No buffer space is available to send the message.
SOCEWOULDBLOCK	The *s* parameter is in nonblocking mode and the data cannot be sent without blocking, or the SO_SNDTIMEO option has been set for socket *s* and the timeout expired before any data was sent.
SOCENOTCONN	The socket is not connected.
SOCEDESTADDRREQ	Destination address required.

Related Calls

getsockopt()
readv()
recv()
recvfrom()
recvmsg()
select()
send()
sendmsg()
setsockopt()
shutdown()
sock_errno()
socket()
writev()

----

setsockopt()

The socket call sets options associated with a socket.

Syntax

```
#include <types.h>
#include <sys\socket.h>
int setsockopt(s, level, optname, optval, optlen)
int s;
int level;
int optname;
char *optval;
int optlen;
```

Parameters

s
Socket descriptor.

level
Specifies which option level is being set.

optname
Name of a specified socket option.

optval
Pointer to the option data.

optlen
Length of the option data.

Description

This call provides an application program with the means to control a socket communication. The setsockopt() call can be used to set options associated with a socket, such as enabling debugging at the socket or protocol level, controlling timeouts, or permitting socket data broadcasts. Options can exist at the socket or the protocol level; options are always present at the highest socket level. When setting socket options, the level of the option and the name of the option must be specified. The following table lists the supported levels:

Supported Levels

Supported Level	#define in
SOL_SOCKET	<SYS\SOCKET.H>
IPPROTO_IP	<NETINET\IN.H>
IPPROTO_TCP	<NETINET\IN.H>
NBPROTO_NB	<NETNB\NB.H>

The *optval* and *optlen* parameters are used to pass data used by the particular set command. The *optval* parameter points to a buffer containing the data needed by the set command. The *optval* parameter is optional and if data is not needed by the command, can be set to the NULL pointer. The *optlen* parameter must be set to the size of the data or data type pointed to by *optval*. For socket options that are toggles, the option is enabled if *optval* is nonzero and disabled if *optval* is 0.

The following tables list the supported options for setsockopt() at each level (SOL_SOCKET, IPPROTO_IP, IPPROTO_TCP). Detailed descriptions of the options follow each table.

Supported setsockopt() Socket Options for SOL_SOCKET

Option Name	Description	Domains(*)	Data Type	Boolean or Value
SO_BROADCAST	Allow sending of broadcast messages	I, N	int	Boolean
SO_DEBUG	Turn on recording of debugging information	I, L	int	Boolean
SO_DONTROUTE	Bypass routing tables	I	int	Boolean
SO_KEEPALIVE	Keep connections alive	I	int	Boolean
SO_LINGER	Linger on close if data present	I	struct linger	Value
SO_L_BROADCAST	Limited broadcast sent on all interfaces	I	int	Boolean
SO_OOBINLINE	Leave received OOB data in-line	I	int	Boolean
SO_RCVBUF	Receive buffer size	I, L, N	int	Value

SO_RCVLOWAT	Receive low watermark	I, L	int	Value
SO_RCVTIMEO	Receive timeout	I, L	struct timeval	Value
SO_REUSEADDR	Allow local address reuse	I, N	int	Boolean
SO_REUSEPORT	Allow local address and port reuse	I	int	Boolean
SO_SNDBUF	Send buffer size	I, L, N	int	Value
SO_SNDLOWAT	Send low watermark	I, L	int	Value
SO_SNDTIMEO	Send timeout	I, L	struct timeval	Value
SO_USELOOPBACK	Bypass hardware when possible	I	int	Value

Table Note (*) This column specifies the communication domains to which this option applies: I for internet, L for Local IPC, and N for NetBIOS.

The following options are recognized for SOL_SOCKET:

Option	**Description**
SO_BROADCAST	(Datagram sockets only.) Enables broadcasting of messages. When this option is enabled, the application can send broadcast messages over *s*, if the interface specified in the destination supports broadcasting of packets.
SO_DEBUG	Enables recording of debug information for a socket. This options is a prerequisite for tracing TCP debug information through the inetdbg utility. For more information enter `inetcfg -?`.
SO_DONTROUTE	Enables the socket to bypass the routing of outgoing messages. When this option is enabled, it causes outgoing messages to bypass the standard routing algorithm and be directed to the appropriate network interface according to the network portion of the destination address. When this option is enabled, packets can be sent only to directly connected networks (networks for which this host has an interface).
SO_KEEPALIVE	(Stream sockets only.) Enables the socket to send keepalive packets that will keep the connection alive. TCP uses a timer called the keepalive timer. This timer is used to monitor idle connections that might have been disconnected because of a peer crash or timeout. When this option is enabled, a keepalive packet is periodically sent to the peer. This is mainly used to allow servers to close connections that are no longer active as a result of clients going away without properly closing connections.
SO_LINGER	(Stream sockets only.) Enables the socket to linger on close if data is present. When this option is enabled and there is unsent data present when soclose() is called, the calling application is blocked during the soclose() call until the data is transmitted or the connection has timed out. When this option is disabled, the soclose() call returns without blocking the caller, and TCP waits to try to send the data. Although the data transfer is usually successful, it cannot be guaranteed because TCP waits only a finite amount of time to send the data. The *optval* parameter points to a linger structure, defined in <SYS\SOCKET.H>: **Field** / **Description** *l_onoff* — Option on/off *l_linger* — Linger time The *l_onoff* field is set to zero if the SO_LINGER option is being disabled. A nonzero value enables the option. The *l_linger* field specifies the amount of time in seconds to linger on close. A value of zero will cause soclose() to wait until the disconnect completes.
SO_L_BROADCAST	Sets limited broadcast sent on all interfaces.

SO_OOBINLINE	(Stream sockets only.) Enables the socket to receive out-of-band data. Out-of-band data is a logically separate data path using the same connection as the normal data path. When this option is enabled, it causes out-of-band data to be placed in the normal data input queue as it is received, making it available to recv(), and recvfrom(), without having to specify the MSG_OOB flag in those calls. When this option is disabled, it causes out-of-band data to be placed in the priority data input queue as it is received, making it available to recv() and recvfrom() only by specifying the MSG_OOB flag in those calls.
SO_RCVBUF	Sets buffer size for input. This option sets the size of the receive buffer to the value contained in the buffer pointed to by *optval*. This allows the buffer size to be tailored for specific application needs, such as increasing the buffer size for high-volume connections. Use inetcfg -g tcprwinsize to see the default and maximum receive socket buffer sizes for stream (TCP) sockets or raw sockets. Use inetcfg -g udprwinsize to see the default and maximum receive socket buffer sizes for UDP sockets.
SO_RCVLOWAT	Sets the receive low watermark.
SO_RCVTIMEO	Sets the receive timeout. The *optval* parameter is a pointer to a timeval structure, which is defined in <SYS\TIME.H>. See Example of recvmsg() Call for an example of setting the socket timeout option.
SO_REUSEADDR	(Stream and datagram sockets only.) Enables a socket to reuse a local address. When this option is enabled, local addresses that are already in use can be bound. This alters the normal algorithm used in the bind() call. The system checks at connect time to be sure that no local address and port have the same foreign address and port. The error SOCEADDRINUSE is returned if the association already exists. Multicast applications must set this socket option if they want to join the same Class D IP address and port for sending and receiving multicast packets.
SO_REUSEPORT	Specifies that the rules used in validating addresses supplied by a bind subroutine should allow reuse of a local port/address combination. Each binding of the port/address combination must specify the SO_REUSEPORT socket option
SO_SNDBUF	Sets the send buffer size. This option sets the size of the send buffer to the value contained in the buffer pointed to by *optval*. This allows the send buffer size to be tailored for specific application needs, such as increasing the buffer size for high-volume connections. Use inetcfg -g tcpswinsize to see the default and maximum send socket buffer sizes for stream (TCP) sockets or raw sockets. Use inetcfg -g udpswinsize to see the default and maximum send socket buffer sizes for UDP sockets.
SO_SNDLOWAT	Sets the send low watermark.
SO_SNDTIMEO	Sets the send timeout. The *optval* parameter is a pointer to a timeval structure, which is defined in <SYS\TIME.H>. See Example of recvmsg() Call for an example of setting the socket timeout option.
SO_USELOOPBACK	Bypasses hardware when possible.

Supported setsockopt() Socket Options for IPPROTO_IP

Option Name	Description	Data Type	Boolean or Value
IP_ADD_MEMBERSHIP	Join a multicast group	struct ip_mreq	Value
IP_DROP_MEMBERSHIP	Leave a multicast group	struct ip_mreq	Value
IP_HDRINCL	Header is included with data	int	Boolean
IP_MULTICAST_IF	Default interface for outgoing multicasts	struct in_addr	Value
IP_MULTICAST_LOOP	Loopback of outgoing	uchar	Boolean

	multicast		
IP_MULTICAST_TTL	Default TTL for outgoing multicast	uchar	Value
IP_OPTIONS	IP options	char *	Value
IP_RECVDSTADDR	Queueing IP destination address	int	Boolean
IP_RECVTRRI	Queueing token ring routing information	int	Boolean
IP_RETOPTS	IP options to be included in outgoing datagrams	char *	Value
IP_TOS	IP type of service for outgoing datagrams	int	Value
IP_TTL	IP time to live for outgoing datagrams	int	Value

The following options are recognized for IPPROTO_IP:

Option	**Description**
IP_ADD_MEMBERSHIP	Used to join a multicast group. There can be 20 groups per socket, and the maximum number of groups for the entire OS/2 TCP/IP system is 320. A multicast packet is delivered to a socket if it has joined the same group on the same interface on which the packet arrived. More than one socket can bind() on a multicast (Class D IP) address and a common port, such as when two clients want to receive the same multicast packet. These sockets must set the SO_REUSEADDR socket option.
IP_DROP_MEMBERSHIP	Used to leave a multicast group.
IP_HDRINCL	(Raw sockets only.) When set, the IP header is included with the data received on the socket.
IP_MULTICAST_IF	Sets the default interface for outgoing multicasts.
IP_MULTICAST_LOOP	This option is used for sending multicast packets. It enables or disables loopback of outgoing multicast packets and is enabled by default. If loopback is disabled, outgoing multicast packets will not loopback in this system; this means that other applications running in this system will not receive outgoing multicast packets even if they have joined the same multicast group.
IP_MULTICAST_TTL	Sets the default TTL for outgoing multicast packets.
IP_OPTIONS	Sets IP options. Same as IP_RETOPTS. See Example of IP_RETOPTS Socket Call for how to use IP_RETOPS.
IP_RECVDSTADDR	(UDP only) Sets the queueing IP destination address. See Example of recvmsg() Call for the way to get this information through recvmsg() call.
IP_RECVTRRI	(UDP packets on token ring only.) Sets queueing token ring routing information. See Example of recvmsg() Call for the way to get this information through recvmsg() call.
IP_RETOPTS	Sets the IP options to be included in outgoing datagrams. See Example of IP_RETOPTS Socket Call for how to use IP_RETOPS.
IP_TOS	Sets the IP type of service for outgoing datagrams.
IP_TTL	Sets the IP time to live value for outgoing datagrams.

Supported setsockopt() Socket Options for IPPROTO_TCP

Option Name	Description	Data Type	Boolean or Value
TCP_CC	Connection count flag	int	Boolean

TCP_MAXSEG	Maximum segment size	int	Value
TCP_MSL	TCP MSL value	int	Value
TCP_NODELAY	Do not delay sending to coalesce packets	int	Boolean
TCP_TIMESTMP	TCP timestamp flag	int	Boolean
TCP_WINSCALE	Window scale flag	int	Boolean

The following options are recognized for IPPROTO_TCP:

Option	Description
TCP_CC	(T/TCP only.) Enables or disables the connection count function status flag (RFC 1644).
TCP_MAXSEG	Sets the maximum segment size.
TCP_MSL	Sets the TCP Maximum Segment Lifetime (MSL) value.
TCP_NODELAY	(Stream sockets only.) Setting on disables the buffering algorithm so that the client's TCP sends small packets as soon as possible. This often has no performance effects on LANs, but can degrade performance on WANs.
TCP_TIMESTMP	(T/TCP only.) Enables or disables the timestamp function status flag (RFC 1323). For more information about high performance, see TCP Extensions for High Performance (RFC 1323).
TCP_WINSCALE	(T/TCP only.) Enables or disables the window scale function status flag (RFC 1323). For more information about high performance, see TCP Extensions for High Performance (RFC 1323).

Supported setsockopt() Socket Options for NBPROTO_NB

Option Name	Description	Data Type	Boolean or Value
NB_DGRAM_TYPE	Type of datagrams to receive	int	Value

The following option is recognized for NBPROTO_NB:

Option	Description
NB_DGRAM_TYPE	(Datagram sockets only.) Sets type of datagrams to be received on the socket. The possible values are: NB_DGRAM — The socket is to receive normal (unicast) datagrams only. NB_BROADCAST — The socket is to receive broadcast datagrams only. NB_DGRAM_ANY — The socket can receive both normal or broadcast datagrams. This option can be changed at any time.

Return Values

The value 0 indicates success; the value -1 indicates an error. You can get the specific error code by calling sock_errno() or psock_errno().

sock_errno() Value	Description
SOCEADDRINUSE	The address is already in use.
SOCENOTSOCK	The *s* parameter is not a valid socket descriptor.

SOCEFAULT	Using *optval* and *optlen* parameters would result in an attempt to access memory outside the caller's address space.
SOCENOPROTOOPT	The *optname* parameter is unrecognized.
SOCEINVAL	Invalid argument.
SOCENOBUFS	No buffer space is available.

Examples

The following are examples of the setsockopt() call. See getsockopt() for examples of how the options are queried.

```
int rc;
int s;
int optval;
struct linger lstruct;
/* extracted from sys/socket.h */
int setsockopt(int s, int level, int optname, char *optval, int optlen);
...
/* I want out of band data in the normal input queue */
optval = 1;
rc = setsockopt(s, SOL_SOCKET, SO_OOBINLINE, (char *) &optval, sizeof(int));
...
/* I want to linger on close */
lstruct.l_onoff  = 1;
lstruct.l_linger = 100;
rc = setsockopt(s, SOL_SOCKET, SO_LINGER, (char *) &lstruct, sizeof(lstruct));
```

Related Calls

- bind()
- endprotoent()
- getprotobyname()
- getprotobynumber()
- getprotoent()
- getsockopt()
- ioctl()
- setprotoent()
- sock_errno()
- socket()

----

shutdown()

The socket call shuts down all or part of a full-duplex connection.

Syntax

```
#include <sys\socket.h>
int shutdown(s, howto)
int s;
int howto;
```

Parameters

s
: Socket descriptor.

howto
: Condition of the shutdown.

Description

This call shuts down all or part of a full-duplex connection. Since data flows in one direction are independent of data flows in the other direction, the shutdown call allows you to independently stop data flow in either direction or all data flows with one API call. For example, you may want to stop the sender(s) from sending data to you, but you still want to send data.

Using the shutdown() call is optional.

The *howto* parameter sets the condition for shutting down the connection to socket *s*. It can be set to one of the following:

- 0 - no more data can be received on socket *s*.
- 1 - no more output to be allowed on the socket *s*.
- 2 - no more data can be sent or received on socket *s*.

Note: In the NetBIOS domain, the shutdown() call has no effect. When called, shutdown() will return a successful return code, but no shutdown occurs.

Return Values

The value 0 indicates success; the value -1 indicates an error. You can get the specific error code by calling sock_errno() or psock_errno().

Error Code	**Description**
SOCENOTSOCK	The *s* parameter is not a valid socket descriptor.
SOCEINVAL	The *howto* parameter was not set to one of the valid values.

Related Calls

accept()
connect()
getsockopt()
readv()
recv()
recvfrom()
recvmsg()
select()
send()
sendto()
setsockopt()
sock_errno()
soclose()
socket()
writev()

----

so_cancel()

The socket call cancels a pending blocking sockets API call on a socket.

Syntax

```
#include <types.h>
#include <sys\socket.h>
int so_cancel (s)
int s;
```

Parameters

s
Socket descriptor.

Description

The so_cancel() call is used in multithreaded applications where one thread needs to 'wake up' another thread which is blocked in a sockets API call.

The thread that has been 'awakened' will return a value of -1 from the sockets API call, and the error will be set to SOCEINTR. If multiple threads are blocked on the same socket and so_cancel() is issued for that socket, only one of the threads will be 'awakened.'

When a socket is in blocking mode, if no threads are blocking on the socket when so_cancel() is issued, the next sockets API call to be issued on that socket will return SOCEINTR. When a socket is in nonblocking mode and no threads are blocking on the socket when so_cancel() is issued, the next call to select() that includes the socket will return SOCEINTR.

Return Values

The value 0 indicates success; the value -1 indicates an error. You can get the specific error code by calling sock_errno() or psock_errno().

Error Code	**Description**
SOCENOTSOCK	The *s* parameter is not a valid socket descriptor.

----

sock_errno()

The socket call returns error code set by a socket call.

Syntax

```
#include <sys/socket.h>
int sock_errno()
```

Description

The sock_errno() call returns the last error code set by a socket call in the current thread. Subsequent socket API calls do not reset this error code.

Related Calls

- ioctl()
- os2_ioctl()
- psock_errno()

----

socket()

The socket call creates an endpoint for communication and returns a socket descriptor representing the endpoint.

Syntax

```
#include <types.h>
#include <sys\socket.h>
int socket(domain, type, protocol)
int domain;
int type;
int protocol;
```

Parameters

domain
Communication domain requested.

type
Type of socket created.

protocol
Protocol requested.

Description

This call creates an endpoint for communication and returns a socket descriptor representing the endpoint. Each socket type provides a different communication service.

Sockets are deallocated with the soclose() call.

The *domain* parameter specifies a communications domain where communication is to take place. This parameter specifies the protocol family which is used.

Protocol Family	Description
PF_OS2 or PF_UNIX	Use addresses in the Local IPC format which take the form of OS/2 Warp file and path names.
PF_INET	Use addresses in the internet address format.
PF_NETBIOS or PF_NB	Use addresses in the NetBIOS address format.
PF_ROUTE	A routing socket can be created with PF_ROUTE as the domain name and SOCK_RAW as the type. A process can use a routing socket to send and receive routing messages.

The *type* parameter specifies the type of socket created. These socket type constants are defined in the <SYS\SOCKET.H> header file. See Socket Types for additional details. The types supported are:

Type	Description
SOCK_STREAM	Provides sequenced, two-way byte streams that are reliable and connection-oriented. It supports a mechanism for out-of-band data. Stream sockets are supported by the internet (PF_INET) communication domain and local IPC (PF_OS2, PF_UNIX, or PF_LOCAL).
SOCK_DGRAM	Provides datagrams, which are connectionless messages of a fixed length whose reliability is not guaranteed. Datagrams can be received out of order, lost, or delivered multiple times. Datagram sockets are supported by the internet (PF_INET), local IPC (PF_OS2, PF_UNIX, or PF_LOCAL), and NetBIOS (PF_NETBIOS or PF_NB) communication domains.
SOCK_RAW	Provides the interface to internal protocols (such as IP and ICMP). Raw sockets are supported by the internet (PF_INET) communication domain.
SOCK_SEQPACKET	Provides sequenced byte streams that are reliable and connection-oriented. Data is sent without error or duplication and is received in the same order as it was sent. Sequenced packet sockets are supported by the NetBIOS (PF_NETBIOS or PF_NB) communication domain.

The *protocol* parameter specifies a particular protocol to be used with the socket. If the protocol field is set to 0 (default), the system selects the default protocol number for the domain and socket type requested. Default and valid protocol number-protocol family combinations are in the section Socket Protocol Families. The getprotobyname() call can be used to get the protocol number for a protocol with a well-known name.

Return Values

A non-negative socket descriptor return value indicates success. The return value -1 indicates an error. You can get the specific error code by calling sock_errno() or psock_errno().

sock_errno() Value	Description
SOCEMFILE	The maximum number of sockets are currently in use.
SOCEPROTONOSUPPORT	The *protocol* is not supported in the specified *domain* or the *protocol* is not supported for the specified socket *type*.
SOCEPFNOSUPPORT	The *protocol family* is not supported.
SOCESOCKTNOSUPPORT	The *socket type* is not supported.

Examples

Following are examples of the socket() call.

```
int s;
struct protoent *p;
struct protoent *getprotobyname(char *name);
int socket(int domain, int type, int protocol); /* extracted from sys/socket.h */
...
/* Get stream socket in internet domain with default protocol */
s = socket(PF_INET, SOCK_STREAM, 0);
...
/* Get raw socket in internet domain for ICMP protocol */
p = getprotobyname("icmp");
s = socket(PF_INET, SOCK_RAW, p->p_proto);
```

Related Calls

accept()
accept_and_recv()
bind()
connect()
getsockname()
getsockopt()
ioctl()
listen()
os2_ioctl()
os2_select()
readv()
recv()
recvfrom()
recvmsg()
select()
send()
sendmsg()
sendto()
setsockopt()
shutdown()
sock_errno()
soclose()
writev()

----

soclose()

The socket call shuts down a socket and frees resources allocated to the socket.

Syntax

```
#include <types.h>
#include <unistd.h>
int soclose(s)
int s;
```

Parameters

s
Socket descriptor.

Description

This call shuts down the socket associated with the socket descriptor *s*, and frees resources allocated to the socket. If *s* refers to a connected socket, the connection is closed.

If the SO_LINGER socket option is enabled (see setsockopt() for additional information), then the task will try to send any queued data. If the SO_LINGER socket option is disabled, then the task will flush any data queued to be sent.

Return Values

The value 0 indicates success; the value -1 indicates an error. You can get the specific error code by calling sock_errno() or psock_errno().

Error Code	**Description**
SOCENOTSOCK	The *s* parameter is not a valid socket descriptor.
SOCEALREADY	The socket *s* is marked nonblocking, and a previous connection attempt has not completed.

Related Calls

accept()
getsockopt()
setsockopt()
sock_errno()
socket()

----

sysctl()

The sysctl() call performs special operations on the TCP/IP stack. Unlike getsockopt() or setsockopt(), sysctl() accesses and modifies systemwide parameter values for the entire TCP/IP stack.

Syntax

```
#include <types.h>
#include <netinet\in.h>
#include <netinet\ip_var.h>
#include <sys\socket.h>
#include <sys\sysctl.h>
int sysctl(mib, namelen, oldp, oldenp, newp, newlen)
int *mib;
u_int namelen;
void *oldp;
size_t newlen;
size_t *oldlenp;
void   *newp;
```

Parameters

mib
Array of integers consisting of command, protocol, and control functions.

namelen
Length of *mib* array.

oldp
Data pointer or xxx_ctl structure pointer pointing to data to be sent.

oldenp
Pointer to length of *oldp* .

newp
Data pointer or xxx_ctl structure pointer pointing to location where data is to be received.

newlen
Length of *newp* .

Description

The sysctl() call is functionally similar to the ioctl() call but does not need a socket to carry the options to and from the stack.

The sysctl() function retrieves stack parameters and allows them to be set. The information available from sysctl() consists of integers, strings, and tables. Unless explicitly noted below, sysctl() returns a consistent snapshot of the data requested. Consistency is achieved by locking the destination buffer into memory so that the data may be copied without blocking.

Calls to sysctl() are serialized to avoid deadlock. The state is described using a Management Information Base (MIB) style name, listed below, which is a *namelen* length array of integers. The information is copied into the buffer specified by *oldp* . The size of the buffer is given by the location specified by *oldlenp* before the call, and that location gives the amount of data copied after a successful call. If the amount of data available is greater than the size of the buffer supplied, the call supplies as much data as fits in the buffer provided and returns with the error code SOCENOMEM.

If the old value is not desired, *oldp* and *oldlenp* should be set to NULL. The size of the available data can be determined by calling sysctl() with a NULL parameter for *oldp* .

The size of the available data will be returned in the location pointed to by *oldenp* . For some operations, the amount of space may change often. For these operations, the system attempts to round up so that the returned size is large enough for a call to return the data shortly thereafter.

To set a new value, *newp* is set to point to a buffer of length *newlen* from which the requested value is to be taken. If a new value is not to be set, *newp* should be set to NULL and *newlen* set to 0.

An Inetcfg sysctl needs a fifth mib argument, mib[4], to specify the actual inet configuration command.

All route sysctl() calls use another additional argument to be carried in mib[5] for rt_flags.   The old *newp* may be pointing to a single integer or char buffer. Also, there are two special control structures (inetver_ctl and intecfg_ctl) used as *oldp* / *newp* structures. Similarly, for statistics the xxxstat structures should be used.

An application uses the OS2_MEMMAPIO sysctl() call to request the TCP/IP stack to provide kernel memory for performing High Performance Send (HPS). One such call can return up to 60K (as 15 4K buffers) of memory. The calling application provides an array named *oldp* of up to 15 pointers (to char). On return from this call, these pointers point to the 4K buffers. The memory acquired in this way is now owned by the application, and it resides in the address space of this application. As a result, the application is now responsible for the management of this memory from a reusability point of view. Applications can use either semaphores or the OS2_QUERY_MEMMAPIO sysctl() call for this purpose. Typically, before calling the next high performance send (which may use one of these buffers), the application needs to verify that the buffers are free to be reused. Sysctl() supports a maximum of 64 such calls. Thus, the kernel can provide up to 64 times 60K of high speed send memory to an application. An ENOMEM error code is returned to any sysctl() call beyond this limit. A sample of usage of this call for supporting HPS is contained in High Performance Send.

An application uses the OS2_QUERY_MEMMAPIO sysctl() call to verify the reusability of the buffers provided by the kernel through the OS2_MEMMAPIO sysctl() call. This call sends the *oldp* array of pointers which were filled in by the kernel during the OS2_MEMMAPIO sysctl() call. *oldenp* is used to pass the number of 4K buffers referred to in the *oldp* array. If a particular pointer in the *oldp* array is left unchanged on return from this call, that buffer has been freed for reuse. Conversely, if a particular pointer in this array is returned as NULL, this buffer is not yet freed and may not be reused. It is the responsibility of the application to make these checks. Alternatively, an application may use semaphores to manage the reusability of these buffers. The *oldp* array can be passed with any number of HPS buffers in a single call and this number of buffers (*oldenp*) need not be an integer multiple of 15. The HPS buffers in *oldp* need not be arranged in the same order in which they were obtained. Applications should save a copy of the obtained HPS buffer pointers before calling OS2_QUERY_MEMMAPIO, so that the pointers are not lost if the buffers are not available.

Values

The values that are supported for different categories of mib values are listed in the following tables.

The generic mib array has the following structure

mib Index	Description
0	Top Level identifier
1	Protocol Family
2	Protocol
3	Address Family or Control Command
4	Control Command
5	Flags, etc.

The mib[0] Top Level values are:

Value	Description
CTL_KERN	Sockets (kernel) domain.
CTL_NET	Routing domain.
CTL_OS2	Local Interprocess Communication (afos2) domain.

The mib[1] Protocol Family values are:

Value	Description
PF_INET	Internet protocol family.
PF_OS2	LIPC (afos2) protocol family.
PF_ROUTE	Route protocol family.
KERN_HOSTID	

The mib[2] Protocols values are:

Value	Description
IPPROTO_IP	Internet Protocol.
IPPROTO_TCP	Transmission Control Protocol.
IPPROTO_UDP	User Datagram Protocol.

The mib[3] Control Command values for inetcfg are:

Value	Description
IPCTL_INETCFG	IP inet configuration.
TCPCTL_INETCFG	TCP inet configuration.
UDPCTL_INETCFG	UDP inet configuration.
LIPCCTL_INETCFG	LIPC inet configuration.

The following table is an overview of the sysctl() calls structure, with links to the descriptions of the mib values. The table is the calling tree to get to individual leaves, which are the supported mib values. You read the table by looking on the right-hand side for the leaf for the value you are seeking, then taking the link to the table that defines that value and describes the function of the value. For example, the first leaf, KERNCTL_INETVER, will get you to the table for mib[0]=CTL_KERN, mib[1]=KERN_HOSTID, and mib[2]=IPPROTO_IP.

```
       mib numbers
[0] [1] [2] [3] [4] [5]
CTL_KERN
....KERN_HOSTID
........IPPROTO_IP
............KERNCTL_INETVER on page mibs for INET Version (sockets.sys).
CTL_OS2
....PF_OS2
........IPPROTO_IP
............LIPCCTL_INETVER on page mibs for INET Version (afos2.sys).
............LIPCCTL_INETCFG
................LIPCCTL_DG_RECVSPACE on page mibs for afos2 inetconfig.
................LIPCCTL_DG_SENDSPACE on page mibs for afos2 inetconfig.
................LIPCCTL_ST_RECVSPACE on page mibs for afos2 inetconfig.
................LIPCCTL_ST_SENDSPACE on page mibs for afos2 inetconfig.
....PF_INET
........IPPROTO_IP
............OS2_MEMMAPIO on page mibs for High Performance Memory.
............OS2_QUERY_MEMMAPIO on page mibs for High Performance Memory.
CTL_NET
....PF_INET
........IPPROTO_TCP
............TCPCTL_INETCFG
................TCPCTL_CC on page mibs for TCPCTL inetconfig.
................TCPCTL_KEEPCNT on page mibs for TCPCTL inetconfig.
................TCPCTL_LINGERTIME on page mibs for TCPCTL inetconfig.
```

```
................TCPCTL_MSL on page mibs for TCPCTL inetconfig.
................TCPCTL_MTU on page mibs for TCPCTL inetconfig.
................TCPCTL_REALSLOW on page mibs for TCPCTL inetconfig.
................TCPCTL_TCPRWIN on page mibs for TCPCTL inetconfig.
................TCPCTL_TCPSWIN on page mibs for TCPCTL inetconfig.
................TCPCTL_TIMESTMP on page mibs for TCPCTL inetconfig.
................TCPCTL_TTL on page mibs for TCPCTL inetconfig.
................TCPCTL_WINSCALE on page mibs for TCPCTL inetconfig.
............TCPCTL_MSSDFLT on page mibs for TCPCTL.
............TCPCTL_RTTDFLT on page mibs for TCPCTL.
............TCPCTL_STATS on page mibs for TCPCTL.
........IPPROTO_UDP
............UDPCTL_INETCFG
................UDPCTL_TTL on page mibs for UDPCTL inetconfig.
................UDPCTL_UDPRWIN on page mibs for UDPCTL inetconfig.
................UDPCTL_UDPSWIN on page mibs for UDPCTL inetconfig.
............UDPCTL_CHECKSUM on page mibs for UDPCTL.
............UDPCTL_STATS on page mibs for UDPCTL.
........IPPROTO_IP
............IPCTL_INETVER on page mibs for IPCTL.
............IPCTL_FORWARDING on page mibs for IPCTL.
............IPCTL_SENDREDIRECTS on page mibs for IPCTL.
............IPCTL_INETCFG
................FRAGCTL_TTL on page mibs for IPCTL inetconfig.
................ICMPCTL_TTL on page mibs for IPCTL inetconfig.
................IPCTL_ARPTKILLC on page mibs for IPCTL inetconfig.
................IPCTL_ARPTKILLI on page mibs for IPCTL inetconfig.
................IPCTL_FIREWALL on page mibs for IPCTL inetconfig.
................IPCTL_FORWARD on page mibs for IPCTL inetconfig.
................IPCTL_MULTIDEFROUTES on page mibs for IPCTL inetconfig.
................IPCTL_SYNATTACK on page mibs for IPCTL inetconfig.
........IPPROTO_ICMP
............ICMPCTL_ECHOREPL on page mibs for ICMPCTL.
............ICMPCTL_MASKREPL on page mibs for ICMPCTL.
............ICMPCTL_STATS on page mibs for ICMPCTL.
....PF_ROUTE
........0 (wildcard)
............0 (wildcard)
................NET_RT_DUMP on page mibs for ROUTE.
....................NET_RT_LLINFO on page mibs for ROUTE with Flags.
................NET_RT_FLAGS on page mibs for ROUTE.
................NET_RT_IFLIST on page mibs for ROUTE.
[0] [1] [2] [3] [4] [5]
       mib numbers
```

Return Values

The requested values are returned in the *newp* parameter. The sysctl() call itself returns the number of bytes copied, if the call is successful. Otherwise, -1 is returned and the errno value is set appropriately.

Error Code	Description
SOCENOPROTOOPT	The protocol specified in mib[1] is not valid.
SOCENOTDIR	The length specified in *namelen* is not valid.
SOCEOPNOTSUPP	The option specified in mib[3] is not supported.
SOCEINVAL	Insufficient mib parameters were supplied.
SOCENOMEM	Memory allocation failed. This error value is returned by High Performance Send (HPS) sysctl (OS2_MEMMAPIO).
SOCEPERM	This parameter cannot be set, it can only be retrieved. Therefore, *newp* must be set to NULL.

Related Calls

ioctl()
os2_ioctl()

----

mibs for INET Version (sockets.sys)

This table shows the mib[3] values supported for

```
mib[0]=CTL_KERN
mib[1]=KERN_HOSTID
mib[2]=0
```

mib[3] Values	Data Type	Description
KERNCTL_INETVER	struct inetver_ctl	Get (no set) the SOCKETS.SYS version number.

----

mibs for INET Version (afos2.sys)

This table shows the mib[3] values supported for

```
mib[0]=CTL_OS2
mib[1]=PF_OS2
mib[2]=0
```

mib[3] Values	Data Type	Description
LIPCCTL_INETVER	struct inetvers_ctl	Get (no set) the AFOS2.SYS (LIPC) version number.

----

mibs for afos2 inetconfig

This table shows the mib[4] values supported for

```
mib[0]=CTL_OS2
mib[1]=PF_OS2
mib[2]=0
mib[3]=LIPCCTL_INETCFG
```

mib[4] Values	Data Type	Description
LIPCCTL_DG_RECVSPACE	int	Get or set the datagram recieve space.
LIPCCTL_DG_SENDSPACE	int	Get or set the datagram send space.
LIPCCTL_ST_RECVSPACE	int	Get or set the stream receive space.
LIPCCTL_ST_SENDSPACE	int	Get or set the stream send space.

----

mibs for High Performance Memory

This table shows the mib[3] values supported for

```
mib[0]=CTL_OS2
mib[1]=PF_INET
mib[2]=0
```

mib[3] Values	Data Type	Description
OS2_MEMMAPIO	long *	Get (no set) high performance memory.
OS2_QUERY_MEMMAPIO	long *	Get (no set) high performance memory reusability status.

----

mibs for TCPCTL inetconfig

This table shows the mib[4] values supported for

```
mib[0]=CTL_NET
mib[1]=PF_INET
mib[2]=IPPROTO_TCP
mib[3]=TCPCTL_INETCFG
```

mib[4] Values	Data Type	Description
TCPCTL_CC	int	Get or set the CC, CCnew and echo flag on or off.
TCPCTL_KEEPCNT	int	Get or set the number of keepalive probes.
TCPCTL_LINGERTIME	int	Get or set the linger on close time.
TCPCTL_MSL	int	Get or set the TCP maximum segment lifetime value.
TCPCTL_MTU	int	Get or set the path maximum transmission unit (MTU) discovery flag on or off.
TCPCTL_REALSLOW	int	Get or set the real slow timer value for the time wait queue.
TCPCTL_TCPRWIN	int	Get or set the TCP receive window size.
TCPCTL_TCPSWIN	int	Get or set the TCP send window size.
TCPCTL_TIMESTMP	int	Get or set the TCP timestamp flag on or off.
TCPCTL_TTL	int	Get or set the time to live (TTL) for TCP packets.

TCPCTL_WINSCALE	int	Get or set the window scale (fat pipe) flag on or off.

----

mibs for TCPCTL

This table shows the mib[3] values supported for

```
mib[0]=CTL_NET
mib[1]=PF_INET
mib[2]=IPPROTO_TCP
```

mib[3] Values	Data Type	Description
TCPCTL_MSSDFLT	int	Get or set the TCP maximum segment size (MSS) default.
TCPCTL_RTTDFLT	int	Get or set the round trip time (RTT) default.
TCPCTL_STATS	struct tcpstat	Get (no set) the TCP statistics.

----

mibs for UDPCTL inetconfig

This table shows the mib[4] values supported for

```
mib[0]=CTL_NET
mib[1]=PF_INET
mib[2]=IPPROTO_UDP
mib[3]=UDPCTL_INETCFG
```

mib[4] Values	Data Type	Description
UDPCTL_TTL	int	Get or set the time to live (TTL) for UDP packets.
UDPCTL_UDPRWIN	int	Get or set the UDP receive window size.
UDPCTL_UDPSWIN	int	Get or set the UDP send window size.

----

mibs for UDPCTL

This table shows the mib[3] values supported for

```
mib[0]=CTL_NET
mib[1]=PF_INET
mib[2]=IPPROTO_UDP
```

mib[3] Values	Data Type	Description
UDPCTL_CHECKSUM	int	Get or set the UDP checksum computing on or off.
UDPCTL_STATS	struct udpstat	Get (no set) the UDP statistics.

----

mibs for IPCTL

This table shows the mib[3] values supported for

```
mib[0]=CTL_NET
mib[1]=PF_INET
mib[2]=IPPROTO_IP
```

mib[3] Values	Data Type	Description
IPCTL_INETVER	struct inetvers_ctl	Get (no set) the AFINET.SYS version number.
IPCTL_FORWARDING	int	Get or set the IP forwarding flag on or off.
IPCTL_SENDREDIRECTS	int	Get set the Send Redirects flag on or off.

----

mibs for IPCTL inetconfig

This table shows the mib[4] values supported for

```
mib[0]=CTL_NET
mib[1]=PF_INET
mib[2]=IPPROTO_IP
mib[3]=IPCTL_INETCFG
```

mib[4] Values	Data Type	Description
FRAGCTL_TTL	int	Get or set the fragment time to live (TTL).

ICMPCTL_TTL	int	Get or set the ICMP packet time to live (TTL).
IPCTL_ARPTKILLC	int	Get or set the ARP cache completed entry timeout.
IPCTL_ARPTKILLI	int	Get or set the ARP cache incompleted entry timeout.
IPCTL_FIREWALL	int	Get or set the IP firewall flag on or off.
IPCTL_FORWARD	int	Get or set the IP forwarding flag on or off.
IPCTL_MULTIDEFROUTES	int	Get or set the multiple default routes function on or off.
IPCTL_SYNATTACK	int	Get or set the SYN attack flag on or off.

----

mibs for ICMPCTL

This table shows the mib[3] values supported for

```
mib[0]=CTL_NET
mib[1]=PF_INET
mib[2]=IPPROTO_ICMP
```

mib[3] Values	Data Type	Description
ICMPCTL_ECHOREPL	int	Get or set the ICMP echo flag on or off.
ICMPCTL_MASKREPL	int	Get or set the flag to check if the system should respond to ICMP address mask requests on or off.
ICMPCTL_STATS	struct icmpstat	Get (no set) the ICMP statistics.

----

mibs for ROUTE

This table shows the mib[4] values supported for

```
mib[0]=CTL_NET
mib[1]=PF_ROUTE
mib[2]=0
mib[3]=any valid address family or 0
```

mib[4] Values	Data Type	Description
NET_RT_DUMP	char *	Dump the routing table entries corresponding to the address family specified in mib[5]. If the address family is zero, then all routing tables are returned.
NET_RT_FLAGS	char *	Dump the routing table entries corresponding to the routing flag RTF_xxx specified in mib[5].
NET_RT_IFLIST	char *	Return information for all configured interfaces if mib[5] is zero. A nonzero mib[5] value specifies the index for a particular interface, and interface information for only that interface is returned.

----

mibs for ROUTE with Flags

This table shows the mib[5] values supported for

```
mib[0]=CTL_NET
mib[1]=PF_ROUTE
mib[2]=0
mib[3]=any valid address family of 0
mib[4]=NET_RT_DUMP
```

mib[5] Values	Data Type	Description
NET_RT_LLINFO	int	Dump the routing table corresponding to the address family specified in mib[3]. If the address family is zero, return all route tables. It carries the RFT_xxx flags or interface index.

Examples

The following examples illustrate the sysctl() call.

This example uses the sysctl() call to get the protocol driver version.

```
#include <stdio.h>
#include <types.h>
#include <netinet\in.h>
#include <sys\socket.h>
#include <netinet\ip_var.h>
#include <sys\sysctl.h>
```

```
int main(void)
{
        int mib[4],i;
        unsigned int  oldenp=0, newlen=0;
        struct inetvers_ctl  uap_old, *uap_new;

        mib[0]= CTL_KERN;        /* cmd              */
        mib[1]= KERN_HOSTID;     /* Protocol Family.*/
        mib[2]= IPPROTO_IP;      /* Protocol        */
        mib[3]= KERNCTL_INETVER; /* Control command for sysctl */


        uap_new = NULL;
        oldenp  = sizeof(struct inetvers_ctl);

        if (sysctl(mib,4,(void *)&uap_old,&oldenp,(void *)uap_new,newlen) > 0)
           printf ("   SOCKETS.SYS: %s\n",uap_old.versionstr);

}
```

The next example shows an inetcfg sysctl() call that uses a fifth mib argument to specify the actual inet config command. This example sets then gets the value.

```
#include <stdio.h>
#include <types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/tcp_var.h>
#include <sys/sysctl.h>
void  main(void)
{
        int mib[5];
        unsigned int  oldenp=0, newlen;
        struct inetcfg_ctl uap_old, uap_new;

        mib[0]= CTL_NET;         /* Top level idetifier */
        mib[1]= PF_INET;         /* Protocol Family */
        mib[2]= IPPROTO_TCP;     /* Protocol */
        mib[3]= TCPCTL_INETCFG;  /* Control command for tcp_sysctl */
        mib[4]= TCPCTL_KEEPCNT;  /* Particular Inetcfg cmd for sysctl_inetcfg */

        /* Set the Value in stack */
        uap_new.var_cur_val = 4;  /* Send 4 Keepalive probes, rather than 8 */
        newlen  = sizeof(struct inetcfg_ctl);
        if (sysctl(mib,5,(void *)NULL, &oldenp, (void *)&uap_new, newlen) < 0)
            printf("sysctl failed for requested parameter\n");

        /* Get the Value from stack */
        oldenp  = sizeof(struct inetcfg_ctl);
        if (sysctl(mib,5,(void *)&uap_old, &oldenp, (void *)NULL, 0) < 0)
            printf("sysctl failed for requested parameter\n");
        else
            printf("Current stack parameter value is %d\n",uap_old.var_cur_val);
}
```

This example illustrates a sysctl() call that uses the route mib.

```
#include <stdio.h>
#include <string.h>
#include <types.h>
#include <netinet\in.h>
#include <sys\socket.h>
#include <netinet\ip_var.h>
#include <sys\sysctl.h>
#include <net\route.h>
#include <net\if.h>

void main(void)
{
        size_t   needed;
        int      mib[6];
        char     *buf, *next, *lim;
        struct   rt_msghdr *rtm;

        mib[0] = CTL_NET;
        mib[1] = PF_ROUTE;
        mib[2] = 0;            /* Wildcard Protocol */
        mib[3] = 0;            /* Wildcard Address Family */
        mib[4] = NET_RT_IFLIST;
        mib[5] = 0;            /* All interfaces */

        if (sysctl(mib, 6, NULL, &needed, NULL, 0) < 0)
                psock_errno("route-sysctl-estimate");

        if (needed == 0) {
            printf("no routes defined\n");
            return 0;
        }

        buf = malloc(needed);
        if (sysctl(mib, 6, buf, &needed, NULL, 0) < 0)
                psock_errno("sysctl if table");
        lim  = buf + needed;
        for (next = buf; next < lim; next += rtm->rtm_msglen) {
           rtm = (struct rt_msghdr *)next;
           switch(rtm->rtm_type){
              case RTM_IFINFO:
                { struct if_msghdr *ifm = (struct if_msghdr *) rtm;
                  printf("if#  %d ,flags 0x%x\n",ifm->ifm_index,ifm->ifm_flags);
                }
                  break;
              case RTM_NEWADDR:
                  /* Add code for this and for RTM_DELADDR etc.....*/
                  break;
           } /* switch */
        } /* for */
}
```

----

writev()

The socket call writes data from a set of specified buffers on a socket.

Syntax

```
#include <types.h>
#include <sys/uio.h>
int writev(s, iov, iovcnt)
int s;
struc iovec *iov;
int iovcnt;
```

Parameters

s
Socket descriptor.

iov
Pointer to an array of iovec structures.

iovcnt
Number of iovec structures pointed to by the *iov* parameter. The maximum value is 1024.

Description

This call writes data on a socket with descriptor *s*. The data is gathered from the buffers specified by iov[0]...iov[iovcnt-1]. The iovec structure is defined in <SYS/UIO.H> and contains the following fields:

Field	**Description**
iov_base	Pointer to the buffer
iov_len	Length of the buffer

This call writes *iov_len* bytes of data. If there is not enough available buffer space to hold the socket data to be transmitted and the socket is in blocking mode, writev() blocks the caller until additional buffer space becomes available. If the socket is in a nonblocking mode, writev() returns -1 and sets return code to SOCEWOULDBLOCK. See ioctl() for a description of how to set nonblocking mode.

TCP/IP alters *iov_base* and *iov_len* for each element in the input struct iovec array. *iov_base* will point to the next character of the processed (sent or received) data on the original buffer, and *iov_len* will become (input value - processed length). Thus if only partial data has been sent or received and the application expects more data to send or receive, it can pass the same *iovec* back in a subsequent call.

For datagram sockets, this call sends the entire datagram, provided the datagram fits into the protocol buffers. Stream sockets act like streams of information with no boundaries separating data. For example, if an application sends 1000 bytes, each call to this function can send 1 byte, 10 bytes, or the entire 1000 bytes. For a stream socket, an application can place this call in a loop, calling this function until all data has been sent.

Return Values

When successful, the number of bytes of data written is returned. Successful completion does not guarantee the data is written. The return value -1 indicates an error was detected on the sending side of the connection. You can get the specific error code by calling sock_errno() or psock_errno().

sock_errno() Value	**Description**
SOCENOTSOCK	*s* is not a valid socket descriptor.
SOCEFAULT	Using the *iov* and *iovcnt* parameters would result in an attempt to access memory outside the caller's address space.
SOCEINTR	Interrupted system call.
SOCEINVAL	Invalid argument.
SOCENOBUFS	Buffer space is not available to send the message.
SOCEWOULDBLOCK	The *s* parameter is in nonblocking mode and the data cannot be written without blocking, or the SO_SNDTIMEO option has been set for socket *s* and the timeout expired before any data was sent.
SOCEMSGSIZE	The message was too big to be sent as a single datagram.
SOCEDESTADDRREQ	A destination address is required.

Related Calls

connect()
getsockopt()
ioctl()
readv()
recv()
recvfrom()
recvmsg()
select()
send()
sendmsg()
sendto()
setsockopt()
sock_errno()
socket()

TCP/IP Network Utility Routines API

The following table briefly describes each sockets utility function call supported by networking services and identifies where you can find the syntax, parameters, and other appropriate information. The network utility calls described in this section can be used to access services only for the internet communication domain.

TCP/IP Network Utility Routines Quick Reference

Socket Call	Description
dn_comp()	Compresses the expanded domain name
dn_expand()	Expands a compressed domain name to a full domain name
dn_find()	Searches for an expanded name from a list of previously compressed names
dn_skipname()	Skips over a compressed domain name
endhostent()	Closes the ETC\HOSTS file
endnetent()	Closes the ETC\NETWORKS file
endprotoent()	Closes the ETC\PROTOCOL file, which contains information about known protocols
endservent()	Closes the ETC\SERVICES file
gethostbyaddr()	Returns a pointer to information about a host specified by an Internet address
gethostbyname()	Returns a pointer to information about a host specified by a host name
gethostent()	Returns a pointer to the next entry in the ETC\HOSTS file
gethostid()	Returns the unique identifier of the current host
gethostname()	Gets the standard host name for the local host machine
_getlong()	Retrieves long byte quantities
getnetbyaddr()	Returns a pointer to the ETC\NETWORKS file entry that contains the specified network address
getnetbyname()	Returns a pointer to the ETC\NETWORKS file entry that contains the specified network name
getnetent()	Returns a pointer to the next entry in the ETC\NETWORKS file
getprotobyname()	Returns a pointer to the ETC\PROTOCOL file entry specified by a protocol name
getprotobynumber()	Returns a pointer to the ETC\PROTOCOL file entry specified by a protocol number
getprotoent()	Returns a pointer to the next entry in the ETC\PROTOCOL file
getservbyname()	Returns a pointer to the ETC\SERVICES file entry specified by a service name
getservbyport()	Returns a pointer to the ETC\SERVICES file entry specified by a port number
getservent()	Returns a pointer to the next entry in the ETC\SERVICES file
_getshort()	Retrieves short byte quantities
h_errno	Returns the TCP/IP error code
htonl()	Translates byte order from host to

	network for a long integer
htons()	Translates byte order from host to network for a short integer
inet_addr()	Constructs an internet address from character strings representing numbers expressed in standard dotted-decimal notation
inet_lnaof()	Returns the local network portion of an internet address
inet_makeaddr()	Constructs an internet address from a network number and a local address
inet_netof()	Returns the network portion of the internet address in network-byte order
inet_network()	Constructs a network number from character strings representing numbers expressed in standard dotted-decimal notation
inet_ntoa()	Returns a pointer to a string in dotted-decimal notation
ntohl()	Translates byte order from network to host for a long integer
ntohs()	Translates byte order from network to host for a short integer
putlong()	Places long byte quantities into the byte stream
putshort()	Places short byte quantities into the byte stream
Raccept()	Accepts a connection request from a SOCKS server
Rbind()	Binds a local name to the socket
Rconnect()	Requests a connection to a remote host
res_init()	Reads the RESOLV file for the default domain name
res_mkquery()	Makes a query message for the name servers in the internet domain
res_query()	Provides an interface to the server query mechanism
res_querydomain()	Queries the concatenation of *name* and *domain*
res_search()	Makes a query and awaits a response
res_send()	Sends a query to a local name server
rexec()	Allows command processing on a remote host
Rgethostbyname()	Returns a pointer to information about a host specified by a host name
Rgetsockname()	Gets the socket name from the SOCKS server
Rlisten()	Completes the binding necessary for a socket to accept connections and creates a connection request queue for incoming requests
sethostent()	Opens and rewinds the ETC\HOSTS file
setnetent()	Opens and rewinds the ETC\NETWORKS file

`setprotoent()`	`Opens and rewinds the ETC\PROTOCOL file`
`setservent()`	`Opens and rewinds the ETC\SERVICES file`

----

dn_comp()

The dn_comp() call compresses the expanded domain name.

Syntax

```
#include <types.h>
#include <netinet\in.h>
#include <arpa\nameser.h>
#include <resolv.h>
int dn_comp(exp_dn, comp_dn, length, dnptrs, lastdnptr)
u_char *exp_dn;
u_char *comp_dn;
int length;
u_char **dnptrs;
u_char **lastdnptr;
```

Parameters

exp_dn
Pointer to the location of an expanded domain name.

comp_dn
Pointer to an array containing the compressed domain name.

length
Length of the array in bytes pointed to by the *comp_dn* parameter.

dnptrs
Pointer to a list of pointers to previously compressed names in the current message.

lastdnptr
Pointer to the end of the array pointed to by *dnptrs* .

Description

The dn_comp() call compresses a domain name to conserve space. When compressing names, the client process must keep a record of suffixes that have appeared previously. The dn_comp() call compresses a full domain name by comparing suffixes to a list of previously used suffixes and removing the longest possible suffix.

The dn_comp() call compresses the domain name pointed to by the *exp_dn* parameter and stores it in the area pointed to by the *comp_dn* parameter. The dn_comp() call inserts labels into the message as the name is compressed. The dn_comp() call also maintains a list of pointers to the message labels and updates the list of label pointers.

- If the value of the *dnptrs* parameter is null, the dn_comp() call does not compress any names. The dn_comp() call translates a domain name from ASCII to internal format without removing suffixes (compressing). Otherwise, the *dnptrs* parameter is the address of pointers to previously compressed suffixes.

- If the *lastdnptr* parameter is null, the dn_comp() call does not update the list of label pointers.

The dn_comp() call is one of a group of calls that form the resolver. The resolver is a set of functions that perform a translation between domain names and network addresses. Global information used by the resolver calls resides in the _res data structure. The <RESOLV.H> file contains the _res data structure definition. (See The _res Data Structure for more on the _res data structure.)

Return Values

When successful, the dn_comp() call returns the size of the compressed domain name. When unsuccessful, the call returns a value of -1.

Related Calls

- dn_expand()
- dn_find()
- dn_skipname()
- _getlong()
- _getshort()
- putlong()
- putshort()
- res_init()
- res_mkquery()
- res_query()
- res_search()
- res_send()

----

dn_expand()

The dn_expand() call expands a compressed domain name to a full domain name.

Syntax

```
#include <types.h>
#include <netinet\in.h>
#include <arpa\nameser.h>
#include <resolv.h>
int dn_expand(msg, eomorig, comp_dn, exp_dn, length)
u_char *msg;
u_char *eomorig;
u_char *comp_dn;
u_char *exp_dn;
int length;
```

Parameters

msg
: Pointer to the beginning of a message.

eomorig
: Pointer to the end of the original message that contains the compressed domain name.

comp_dn
: Pointer to the compressed domain name.

exp_dn
: Pointer to a buffer that holds the resulting expanded domain name.

length
: Length of the buffer in bytes pointed to by the *exp_dn* parameter.

Description

The dn_expand() call expands a compressed domain name to a full domain name, converting the expanded names to all uppercase letters. A client process compresses domain names to conserve space. Compression consists of removing the longest possible previously occurring suffixes. The dn_expand() call restores a domain name compressed by the dn_comp() call to its full size.

The dn_expand() call is one of a set of calls that form the resolver. The resolver is a group of functions that perform a translation between domain names and network addresses. Global information used by the resolver calls resides in the _res data structure. The <RESOLV.H> file contains the _res data structure definition. (See The _res Data Structure for more on the _res data structure.)

Return Values

When successful, the dn_expand() call returns the size of the expanded domain name. When unsuccessful, the call returns a value of -1.

Related Calls

dn_comp()
dn_find()
dn_skipname()
_getlong()
_getshort()
putlong()
putshort()
res_init()
res_mkquery()
res_query()
res_search()
res_send()

----

dn_find()

The dn_find() call searches for an expanded name from a list of previously compressed names.

Syntax

```
#include <types.h>
#include <netinet\in.h>
#include <arpa\nameser.h>
#include <resolv.h>
int dn_find(exp_dn, msg, dnptrs, lastdnptr)
u_char *exp_dn;
u_char *msg;
u_char **dnptrs;
u_char **lastdnptr;
```

Parameters

exp_dn
Pointer to the expanded name to search for.

msg
Pointer to the start of a messag.e

dnptrs
Pointer to the location of the first name on the list to search, *not* the pointer to the start of the message.

lastdnptr
Pointer to the end of the array pointed to by *dnptrs* .

Description

The dn_find() call is one of a group of calls that form the resolver. The resolver is a set of functions that perform a translation between domain names and network addresses. Global information used by the resolver calls resides in the _res data structure. The <RESOLV.H> file contains the _res data structure definition. (See The _res Data Structure for more on the _res data structure.)

Return Values

When successful, the dn_find() call returns the offset from *msg* . When unsuccessful, the call returns a value of -1.

Related Calls

dn_comp()
dn_expand()
dn_skipname()
_getlong()
_getshort()
putlong()
putshort()
res_init()
res_mkquery()

res_query()
res_search()
res_send()

----

dn_skipname()

The dn_skipname() call skips over the compressed domain name pointed to by the *comp_dn* parameter.

Syntax

```
#include <types.h>
#include <netinet\in.h>
#include <arpa\nameser.h>
#include <resolv.h>
int dn_skipname(comp_dn, eom)
u_char *comp_dn;
u_char *eom;
```

Parameters

comp_dn
Pointer to the compressed domain name.

eom
Pointer to the end of the original message that contains the compressed domain name.

Return Values

When successful, the dn_skipname() call returns the size of the compressed name. When unsuccessful, the call returns a value of -1.

Related Calls

dn_comp()
dn_expand()
dn_find()
_getlong()
_getshort()
putlong()
putshort()
res_init()
res_mkquery()
res_query()
res_search()
res_send()

----

endhostent()

The endhostent() call closes the ETC\HOSTS file, which contains information about known hosts.

Syntax

```
#include <netdb.h>
void endhostent()
```

Description

The endhostent() call closes the ETC\HOSTS file.

When using the endhostent() call in DNS/BIND name service resolution, endhostent() closes the TCP connection which the sethostent() call set up.

Related Calls

gethostbyaddr()
gethostbyname()
gethostent()
sethostent()

----

endnetent()

The endnetent() call closes the ETC\NETWORKS file, which contains information about known networks.

Note: Calls made to the getnetent(), getnetbyaddr(), or getnetbyname() call open the ETC\NETWORKS file.

Syntax

```
#include <netdb.h>
void endnetent()
```

Related Calls

getnetbyaddr()
getnetbyname()
getnetent()
setnetent()

----

endprotoent()

The endprotoent() call closes the ETC\PROTOCOL file, which contains information about known protocols.

Note: Calls made to the getprotoent(), getprotobyname(), or getnetbynumber() call open the ETC\PROTOCOL file.

Syntax

```
#include <netdb.h>
void endprotoent()
```

Related Calls

getprotobyname()
getprotobynumber()
getprotoent()
setprotoent()

----

endservent()

The endservent() call closes the ETC\SERVICES file, which contains information about known services.

Note: Calls made to the getservent(), getservbyname(), or getservbyport() call open the ETC\SERVICES file.

Syntax

```
#include <netdb.h>
void endservent()
```

Related Calls

- endprotoent()
- getprotobyname()
- getprotobynumber()
- getprotoent()
- getservbyname()
- getservbyport()
- getservent()
- setprotoent()
- setservent()

----

gethostbyaddr()

The gethostbyaddr() call returns a pointer to information about a host specified by an internet address.

Syntax

```
#include <netdb.h>
struct hostent *gethostbyaddr(addr, addrlen, addrfam)
char *addr;
int addrlen;
int addrfam;
```

Parameters

addr
Pointer to a 32-bit internet address in network-byte order.

addrlen
Size of *addr* in bytes.

addrfam
Address family supported (AF_INET).

Description

This call resolves the host name through a name server, if one is present. If a name server is not present or cannot resolve the host name, gethostbyaddr() uses the default name services ordering: first it queries DNS/BIND, then it searches the ETC\HOSTS file in sequence until a matching host address is found or an end-of-file (EOF) marker is reached. This search order can be reversed by adding the following statement in your CONFIG.SYS file:

```
SET USE_HOSTS_FIRST=1
```

When using DNS/BIND name service resolution, if the ETC\RESOLV file exists the gethostbyaddr() call queries the domain name server. The gethostbyaddr() call recognizes domain name servers as described in RFC 883.

The gethostbyaddr() call also searches the local ETC\HOSTS file when indicated to do so.

The gethostbyaddr() call returns a pointer to a hostent structure, which contains information obtained from one of the name resolutions services. The hostent structure is defined in the <NETDB.H> file.

Return Values

The return value points to static data that subsequent API calls can modify. This call returns a pointer to a hostent structure for the host address specified on the call and indicates success. A NULL pointer indicates an error.

The <NETDB.H> header file defines the hostent structure and contains the following elements:

Element	**Description**
h_name	Official name of the host
h_aliases	Zero-terminated array of alternative names for the host
h_addrtype	The address family of the network address being returned, always set to AF_INET
h_length	Length of the address in bytes
h_addr	Pointer to the network address of the host

The value of h_errno indicates the specific error.

h_errno Value	**Code**	**Description**
NETDB_INTERNAL	-1	Generic error value. Call sock_errno() or psock_errno() to get a more detailed error code (or error message).
HOST_NOT_FOUND	1	The host specified by the *addr* parameter is not found.
TRY_AGAIN	2	The local server does not receive a response from an authorized server. Try again later.
NO_RECOVERY	3	This error code indicates an unrecoverable error.
NO_DATA	4	The requested *addr* is valid, but does not have an Internet address at the name server.
NO_ADDRESS	4	The requested *addr* is valid, but does not have an Internet address at the name server.

Related Calls

endhostent()
gethostbyname()
gethostent()
inet_addr()
sethostent()

----

gethostbyname()

The gethostbyname() call returns a pointer to information about a host specified by a host name.

Syntax

```
#include <netdb.h>
struct hostent *gethostbyname(name)
char *name;
```

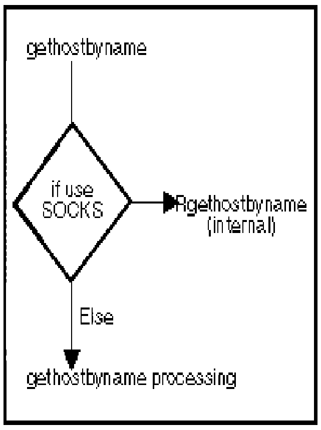
Parameters

name
Pointer to the name of the host being queried.

Description:

The following diagram illustrates gethostbyname() processing:

gethostbyname() Processing



If you are using a SOCKS server, gethostbyname() calls Rgethostbyname(). To avoid having to change your applications should there be changes in SOCKS support, it is recommended you use gethostbyname() rather than Rgethostbyname().

See Socket Secure Support for information about SOCKS.

The gethostbyname() call resolves the host name through a name server, if one is present. If a name server is not present or is unable to resolve the host name, gethostbyname() searches the ETC\HOSTS file in sequence until a matching host name is found or an EOF marker is reached. This search order can be reversed by the following statement in your CONFIG.SYS file:

```
SET USE_HOSTS_FIRST=1
```

Return Values

The return value points to static data that subsequent API calls can modify. This call returns a pointer to a hostent structure for the host address specified on the call and indicates success. A NULL pointer indicates an error.

The <NETDB.H> header file defines the hostent structure and contains the following elements:

Element	**Description**
h_name	Official name of the host
h_aliases	Zero-terminated array of alternative names for the host
h_addrtype	The address family of the network address being returned, always set to AF_INET
h_length	Length of the address in bytes
h_addr	Pointer to the network address of the host

The value of h_errno indicates the specific error.

h_errno Value	**Code**	**Description**
NETDB_INTERNAL	-1	Generic error value. Call sock_errno() or psock_errno() to get a more detailed error code (or error message).
HOST_NOT_FOUND	1	The host specified by the *name* parameter is not found.
TRY_AGAIN	2	The local server does not receive a response from an authorized server. Try again later.
NO_RECOVERY	3	This error code indicates an unrecoverable error.
NO_DATA	4	The requested *name* is valid, but does not have an Internet address at the name server.

`NO_ADDRESS`	`4`	`The requested` *`name`* `is valid, but does not have an Internet address at the name server.`

Related Calls

- endhostent()
- gethostbyaddr()
- gethostent()
- inet_addr()
- sethostent()

----

gethostent()

The gethostent() call returns a pointer to the next entry in the ETC\HOSTS file.

Syntax

```
#include <netdb.h>
struct hostent *gethostent()
```

Description

This call returns a pointer to a hostent structure, which contains the equivalent fields for a host description line in the ETC\HOSTS file. The hostent structure is defined in the <NETDB.H> file.

Return Values

The return value points to static data that subsequent API calls can modify. This call returns a pointer to a hostent structure for the host address specified on the call and indicates success. A NULL pointer indicates an error or EOF.

The <NETDB.H> header file defines the hostent structure and contains the following elements:

Element	Description
h_name	Official name of the host
h_aliases	Zero-terminated array of alternative names for the host
h_addrtype	The address family of the network address being returned, always set to AF_INET
h_length	Length of the address in bytes
h_addr	Pointer to the network address of the host

Related Calls

- endhostent()
- gethostbyaddr()
- gethostbyname()
- sethostent()

----

gethostid()

The gethostid() call returns the unique 32-bit identifier of the current host.

Syntax

```
#include <unistd.h>
u_long gethostid()
```

Return Values

The gethostid() call returns the 32-bit identifier, in host-byte order of the current host, which should be unique across all hosts. This identifier is usually the IP address of the primary interface. The default primary interface is lan0. For a slip only or PPP only configuration, the sl0 or ppp0 is the primary interface. If no primary interface exists, the call returns a hexadecimal of X'FFFFFFFF'.

Related Calls

gethostname()

----

gethostname()

The gethostname() call gets the standard host name for the local host machine.

Syntax

```
#include <unistd.h>
int gethostname(name, namelen)
char *name;
int namelen;
```

Parameters

name
Pointer to a buffer.

namelen
Length of the buffer.

Description

This call copies the standard host name for the local host into the buffer specified by the *name* parameter. The returned name is a null-terminated string. If insufficient space is provided, the returned name is truncated to fit the given space. System host names are limited to 256 characters.

The gethostname() call allows a calling process to determine the internal host name for a machine on a network.

Return Values

The value 0 indicates success; the value -1 indicates an error.

Related Calls

gethostbyaddr()
gethostbyname()
gethostid()

_getlong()

The _getlong() call retrieves long byte quantities.

Syntax

```
#include <sys\types.h>
#include <netinet\in.h>
#include <arpa\nameser.h>
#include <resolv.h>
u_long _getlong (msgp)
```

```
u_char *msgp;
```

Parameters

msgp
: Specifies a pointer into the byte stream.

Description

The _getlong() call gets long quantities from the byte stream or arbitrary byte boundaries.

The _getlong() call is one of a group of calls that form the resolver, a set of functions that resolves domain names. Global information used by the resolver calls is kept in the _res data structure. The <RESOLV.H> file contains the _res structure definition. (See The _res Data Structure for more on the _res data structure.)

Return Values

The _getlong() call returns an unsigned long (32-bit) value.

Related Calls

dn_comp()
dn_expand()
dn_find()
dn_skipname()
_getshort()
putlong()
putshort()
res_init()
res_mkquery()
res_query()
res_search()
res_send()

----

getnetbyaddr()

The getnetbyaddr() call returns a pointer to the ETC\NETWORKS file entry that contains the specified network address.

Syntax

```
#include <netdb.h>
struct netent *getnetbyaddr(net, type)
u_long net;
int type;
```

Parameters

net
: Network address.

type
: Address family supported (AF_INET).

Description

The getnetbyaddr() call searches the ETC\NETWORKS file for the specified network address.

The getnetbyaddr() call retrieves information from the ETC\NETWORKS file using the network address as a search key. The getnetbyaddr() call searches the file sequentially from the start of the file until it encounters a matching net number and type or until it reaches the end of the file.

The getnetbyaddr() call returns a pointer to a netent structure, which contains the equivalent fields for a network description line in the ETC\NETWORKS file. The netent structure is defined in the <NETDB.H> file.

Use the endnetent() call to close the ETC\NETWORKS file.

Return Values

The return value points to static data that subsequent API calls can modify. A pointer to a netent structure indicates success. A NULL pointer indicates an error or EOF.

The netent structure is defined in the <NETDB.H> header file and contains the following elements:

Element	**Description**
n_name	Official name of the network
n_aliases	An array, terminated with a NULL pointer, of alternative names for the network
n_addrtype	The address family of the network address being returned, always set to AF_INET
n_net	Network number, returned in host-byte order

Related Calls

endnetent()
getnetbyname()
getnetent()
setnetent()

----

getnetbyname()

The getnetbyname() call returns a pointer to the ETC\NETWORKS file entry that contains the specified network name.

Syntax

```
#include <netdb.h>
struct netent *getnetbyname(name)
char *name;
```

Parameters

name
Pointer to a network name.

Description

This call searches the ETC\NETWORKS file for the specified network name.

The getnetbyname() call retrieves information from the ETC\NETWORKS file using the *name* parameter as a search key. The getnetbyname() call searches the file sequentially from the start of the file until it encounters a matching net name or until it reaches the end of the file.

The getnetbyname() call returns a pointer to a netent structure, which contains the equivalent fields for a network description line in the ETC\NETWORKS file. The netent structure is defined in the <NETDB.H> file.

Use the endnetent() call to close the ETC\NETWORKS file.

Return Values

The getnetbyname() call returns a pointer to a netent structure for the network name specified on the call. The return value points to static data that subsequent API calls can modify. A pointer to a netent structure indicates success. A NULL pointer indicates an error or EOF.

The netent structure is defined in the <NETDB.H> header file and contains the following elements:

Element	**Description**
n_name	Official name of the network
n_aliases	An array, terminated with a NULL pointer, of alternative names for the network
n_addrtype	The address family of the network address being returned, always set to AF_INET
n_net	Network number, returned in host-byte order

Related Calls

endnetent()

getnetbyaddr()
getnetent()
setnetent()

----

getnetent()

The getnetent() call returns a pointer to the next entry in the ETC\NETWORKS file.

Syntax

```
#include <netdb.h>
struct netent *getnetent()
```

Description

This call, by opening and sequentially reading the ETC\NETWORKS file, returns a pointer to the next entry in the file.

The getnetent() call returns a pointer to a netent structure, which contains the equivalent fields for a network description line in the ETC\NETWORKS file. The netent structure is defined in the <NETDB.H> file.

Use the endnetent() call to close the ETC\NETWORKS file.

Return Values

The getnetent() call returns a pointer to the next entry in the ETC\NETWORKS file. The return value points to static data that subsequent API calls can modify. A pointer to a netent structure indicates success. A NULL pointer indicates an error or EOF.

The netent structure is defined in the <NETDB.H> header file and contains the following elements:

Element	**Description**
n_name	Official name of the network
n_aliases	An array, terminated with a NULL pointer, of alternative names for the network
n_addrtype	The address family of the network address being returned, always set to AF_INET
n_net	Network number, returned in host-byte order

Related Calls

endnetent()
getnetbyaddr()
getnetbyname()
setnetent()

----

getprotobyname()

The getprotobyname() call returns a pointer to the ETC\PROTOCOL file entry specified by a protocol name.

Syntax

```
#include <netdb.h>
struct protoent *getprotobyname(name)
char *name;
```

Parameters

name
Pointer to the specified protocol name string.

Description

This call searches the ETC\PROTOCOL file for the specified protocol name.

The getprotobyname() call retrieves protocol information from the ETC\PROTOCOL file using the *name* parameter as a search key. An application program can use this call to access a protocol name, its aliases, and protocol number.

The getprotobyname() call searches the file sequentially from the start of the file until it encounters a matching protocol name, or until it reaches the end of the file.

The getprotobyname() call returns a pointer to a protoent structure, which contains fields for a line of information in the ETC\PROTOCOL file. The protoent structure is defined in the <NETDB.H> file.

Use the endprotoent() call to close the ETC\PROTOCOL file.

Return Values

The getprotobyname() call returns a pointer to a protoent structure for the network protocol specified on the call. The return value points to static data that subsequent API calls can modify. A pointer to a protoent structure indicates success. A NULL pointer indicates an error or EOF.

The protoent structure is defined in the <NETDB.H> header file and contains the following elements:

Element	**Description**
p_name	Official name of the protocol
p_aliases	Array, terminated with a NULL pointer, of alternative names for the protocol
p_proto	Protocol number

Related Calls

- endprotoent()
- getprotobynumber()
- getprotoent()
- setprotoent()

----

getprotobynumber()

The getprotobynumber() call returns a pointer to the ETC\PROTOCOL file entry specified by a protocol number.

Syntax

```
#include <netdb.h>
struct protoent * getprotobynumber(proto)
int proto;
```

Parameters

proto
: Protocol number.

Description

This call searches the ETC\PROTOCOL file for the specified protocol number.

The getprotobynumber() call retrieves the protocol information from the ETC\PROTOCOL file using the specified *proto* parameter as a search key. An application program can use this call to access a protocol name, its aliases, and protocol number.

The getprotobynumber() call searches the file sequentially from the start of the file until it encounters a matching protocol number, or until it reaches the end of the file.

The getprotobynumber() call returns a pointer to a protoent structure, which contains fields for a line of information in the ETC\PROTOCOL file. The protoent structure is defined in the <NETDB.H> file.

Use the endprotoent() call to close the ETC\PROTOCOL file.

Return Values

The getprotobynumber() call returns a pointer to a protoent structure for the network protocol specified on the call. The return value points to static data that subsequent API calls can modify. A pointer to a protoent structure indicates success. A NULL pointer indicates an error or EOF.

The protoent structure is defined in the <NETDB.H> header file and contains the following elements:

Element	**Description**
p_name	Official name of the protocol
p_aliases	Array, terminated with a NULL pointer, of alternative names for the protocol
p_proto	Protocol number

Related Calls

- endprotoent()
- getprotobyname()
- getprotoent()
- setprotoent()

----

getprotoent()

The getprotoent() call returns a pointer to the next entry in the ETC\PROTOCOL file.

Syntax

```
#include <netdb.h>
struct protoent *getprotoent()
```

Description

This call searches for the next entry in the ETC\PROTOCOL file.

The getprotoent() call retrieves the protocol information from the ETC\PROTOCOL file. An application program can use this call to access a protocol name, its aliases, and protocol number.

The getprotoent() call opens and performs a sequential read of the ETC\PROTOCOL file. The getprotoent() call returns a pointer to a protoent structure, which contains fields for a line of information in the ETC\PROTOCOL file. The protoent structure is defined in the <NETDB.H> file.

Use the endprotoent() call to close the ETC\PROTOCOL file.

Return Values

The getprotoent() call returns a pointer to the next entry in the file, ETC\PROTOCOL. The return value points to static data that subsequent API calls can modify. A pointer to a protoent structure indicates success. A NULL pointer indicates an error or EOF.

The protoent structure is defined in the <NETDB.H> header file and contains the following elements:

Element	**Description**
p_name	Official name of the protocol
p_aliases	Array, terminated with a NULL pointer, of alternative names for the protocol
p_proto	Protocol number

Related Calls

- endprotoent()
- getprotobyname()
- getprotobynumber()
- setprotoent()

----

getservbyname()

The getservbyname() call returns a pointer to the ETC\SERVICES file entry specified by a service name.

Syntax

```
#include <netdb.h>
struct servent *getservbyname(name, proto)
char *name;
char *proto;
```

Parameters

name
Pointer to the service name.

proto
Pointer to the specified protocol.

Description

This call searches the ETC\SERVICES file for the specified service name, which must match the protocol if a protocol is stated.

The getservbyname() call retrieves an entry from the ETC\SERVICES file using the *name* parameter as a search key.

An application program can use this call to access a service, service aliases, the protocol for the service, and a protocol port number for the service.

The getservbyname() call searches the ETC\SERVICES file sequentially from the start of the file until it finds one of the following:

- Matching name and protocol number
- Matching name when the *proto* parameter is set to 0
- End of the file

Upon locating a matching name and protocol, the getservbyname() call returns a pointer to the servent structure, which contains fields for a line of information from the ETC\SERVICES file. The <NETDB.H> file defines the servent structure and structure fields.

Use the endservent() call to close the ETC\SERVICES file.

Return Values

The call returns a pointer to a servent structure for the network service specified on the call. The return value points to static data that subsequent API calls can modify. A pointer to a servent structure indicates success. A NULL pointer indicates an error or EOF.

The servent structure is defined in the <NETDB.H> header file and contains the following elements:

Element	**Description**
s_name	Official name of the service
s_aliases	Array, terminated with a NULL pointer, of alternative names for the service
s_port	Port number of the service
s_proto	Required protocol to contact the service

Related Calls

endprotoent()
endservent()
getprotobyname()
getprotobynumber()
getprotoent()
getservbyport()
getservent()
setprotoent()
setservent()

----

getservbyport()

The getservbyport() call returns a pointer to the ETC\SERVICES file entry specified by a port number.

Syntax

```
#include <netdb.h>
struct servent *getservbyport(port, proto)
int port;
char *proto;
```

Parameters

port
Specified port.

proto
Pointer to the specified protocol.

Description

This call sequentially searches the ETC\SERVICES file for the specified port number, which must match the protocol if a protocol is stated.

The getservbyport() call retrieves an entry from the ETC\SERVICES file using the *port* parameter as a search key.

An application program can use this call to access a service, service aliases, the protocol for the service, and a protocol port number for the service.

The getservbyname() call searches the ETC\SERVICES file sequentially from the start of the file until it finds one of the following:

- Matching protocol and port number
- Matching protocol when the *port* parameter is set to 0
- End of the file

Upon locating a matching protocol and port number-or upon locating a matching protocol only (if the *port* parameter equals 0-the getservbyport() call returns a pointer to the servent structure. This structure contains fields for a line of information from the ETC\SERVICES file. The <NETDB.H> file defines the servent structure and structure fields.

Use the endservent() call to close the ETC\SERVICES file.

Return Values

The getservbyport() call returns a pointer to a servent structure for the port number specified on the call. The return value points to static data that subsequent API calls can modify. A pointer to a servent structure indicates success. A NULL pointer indicates an error or EOF.

The servent structure is defined in the <NETDB.H> header file and contains the following elements:

Element	**Description**
s_name	Official name of the service
s_aliases	Array, terminated with a NULL pointer, of alternative names for the service
s_port	Port number of the service
s_proto	Required protocol to contact the service

Related Calls

endprotoent()
endservent()
getprotobyname()
getprotoent()
getservbyname()
getservent()
setprotoent()
setservent()

----

getservent()

The getservent() call returns a pointer to the next entry in the ETC\SERVICES file.

Syntax

```
#include <netdb.h>
struct servent *getservent()
```

Description

This call searches for the next line in the ETC\SERVICES file. An application program can use the getservent() call to retrieve information about network services and the protocol ports they use.

The getservent() call returns a pointer to a servent structure, which contains fields for a line of information from the ETC\SERVICES file. The servent structure is defined in the <NETDB.H> file.

The ETC\SERVICES file remains open after a call by getservent(). To close the ETC\SERVICES file after each call, use setservent(). Otherwise, use endservent() to close the file.

Return Values

The getservent() call returns a pointer to the next entry in the ETC\SERVICES file. The return value points to static data that subsequent API calls can modify. A pointer to a servent structure indicates success. A NULL pointer indicates an error or EOF.

The servent structure is defined in the <NETDB.H> header file and contains the following elements:

Element	Description
s_name	Official name of the service
s_aliases	Array, terminated with a NULL pointer, of alternative names for the service
s_port	Port number of the service
s_proto	Required protocol to contact the service

Related Calls

endprotoent()
endservent()
getprotobyname()
getprotobynumber()
getprotoent()
getservbyname()
getservbyport()
setprotoent()
setservent()

_getshort()

The _getshort() call retrieves short byte quantities.

Syntax

```
#include <sys\types.h>
#include <netinet\in.h>
#include <arpa\nameser.h>
#include <resolv.h>
u_short _getshort (msgp)
u_char *msgp;
```

Parameters

msgp
Specifies a pointer into the byte stream.

Description

The _getshort() call gets quantities from the byte stream or arbitrary byte boundaries.

The _getshort() call is one of a group of calls that form the resolver, a set of functions that resolve domain names. Global information used by the resolver calls is kept in the _res data structure. The <RESOLV.H> file contains the _res structure definition. (See The _res Data Structure for more on the _res data structure.)

Return Values

The _getshort() call returns an unsigned short (16-bit) value.

Related Calls

dn_comp()
dn_expand()
dn_find()
dn_skipname()
_getlong()
putlong()
putshort()
res_init()
res_mkquery()

----

h_errno

The h_errno variable returns the last name resolution error that occurred in the current thread.

Syntax

```
#include <netdb.h>
int h_errno
```

Description

The h_errno variable returns the last name resolution error that occurred in the current thread. These error codes are set by gethostname(), res_send(), res_query(), res_querydomain(), and res_search().

Return Values

The value -1 indicates an error code is unavailable. A non-zero, non-negative returned value is the TCP/IP error code.

Error Code	**Description**
HOST_NOT_FOUND	The host cannot be found.
TRY_AGAIN	Server failure.
NO_RECOVERY	Non-recoverable error.
NO_DATA	Valid name; no data record of requested type.
NO_ADDRESS	No address; look for MX record (mail exchange record used by DNS).

Related Calls

gethostname()
res_query()
res_querydomain()
res_search()
res_send()

----

htonl()

The socket call translates a long integer from host-byte order to network-byte order.

Syntax

```
#include <types.h>
u_long htonl(a)
u_long a;
```

Parameters

a
: Unsigned long integer to be put into network-byte order.

Description

The htonl() call converts an unsigned long (32-bit) integer from host-byte order to internet network-byte order.

The internet network requires addresses and ports in network standard-byte order. Use the htonl() call to convert the host integer representation of addresses and ports to internet network-byte order.

Return Values

Returns the translated long integer.

Related Calls

htons()
ntohl()
ntohs()

----

htons()

The socket call translates a short integer from host-byte order to network-byte order.

Syntax

```
#include <types.h>
u_short htons(a)
u_short a;
```

Parameters

a
: Unsigned short integer to be put into network-byte order.

Description

The htons() call converts an unsigned short (16-bit) integer from host-byte order to internet network-byte order.

The internet network requires addresses and ports in network standard-byte order. Use the htons() call to convert ports from their host integer representation to internet network-byte order.

Return Values

Returns the translated short integer.

Related Calls

htonl()
ntohl()
ntohs()

----

inet_addr()

The inet_addr() call constructs an internet address from character strings representing numbers expressed in standard dotted-decimal notation.

Syntax

```
#include <arpa\inet.h>
u_long inet_addr(cp)
char *cp;
```

Parameters

cp
A character string in standard dotted-decimal notation

Description

The inet_addr() call converts an ASCII string containing a valid internet address using dotted-decimal notation into an internet address number typed as an unsigned long value. An example of dotted-decimal notation is 120.121.5.123. The inet_addr() call returns an error value if the internet address notation in the ASCII string supplied by the application is not valid.

Note:

Although inet_addr() call and inet_network() call both convert internet addresses in dotted-decimal notation, they process ASCII strings differently. When an application gives the inet_addr() call a string containing an internet address value without a delimiter, the call returns the logical product of the value represented by the string and 0xFFFFFFFF. For any other internet address, if the value of the fields exceeds the previously defined limits, the inet_addr() call returns an error value of -1.

When an application gives the inet_network() call a string containing an internet address value without a delimiter, the inet_network() call returns the logical product of the value represented by the string and 0xFF. For any other internet address, the call returns an error value of -1 if the value of the fields exceeds the previously defined limits.

Sample return values for each call are as follows:

Application string	inet_addr() returns	inet_network() returns
0x1234567890abcdef	0x090abcdef	0x000000ef
0x1234567890abcdef	0xFFFFFFFF (= -1)	0x0000ef00
256.257.258.259	0xFFFFFFFF (= -1)	0x00010203

The ASCII string for the inet_addr() call must conform to the following format:

```
string::= field | field delimited_field^1-3 | delimited_field^1-3
delimited_field::= delimiter field | delimiter
delimiter::= .
field::= 0X | 0x | 0Xhexadecimal* | 0x hexadecimal* | decimal* | 0 octal
hexadecimal::= decimal |a|b|c|d|e|f|A|B|C|D|E|F
decimal::= octal |8|9
octal::= 0|1|2|3|4|5|6|7
```

Notes:

1. *^n* indicates *n* repetitions of a pattern.
2. *^n-m* indicates *n* to *m* repetitions of a pattern.
3. *** indicates 0 or more repetitions of a pattern, up to environmental limits.
4. The Backus Naur form (BNF) description states the space character, if one is used. *Text* indicates text, not a BNF symbol.

The inet_addr() call requires an application to terminate the string with a null terminator (0x00) or a space (0x30). The string is considered invalid if the application does not end it with a null terminator or a space. The call ignores characters trailing a space.

The following describes the restrictions on the field values for the inet_addr() call:

Format	**Field Restrictions (in decimal)**
a	*Value_a* < 4,294,967,296
a.b	*Value_a* < 256; *Value_b* < 16,777,216
a.b.c	*Value_a* < 256; *Value_b* < 256; *Value_c* < 65536
a.b.c.d	*Value_a* < 256; *Value_b* < 256; *Value_c* < 256; *Value_d* < 256

When a four-part address is specified, each part is interpreted as a byte of data and assigned, from left to right, to one of the 4 bytes of an internet address.

When a three-part address is specified, the last part is interpreted as a 16-bit quantity and placed in the two rightmost bytes of the network address. This makes the three-part address format convenient for specifying Class B network addresses as 128.net.host.

When a two-part address is specified, the last part is interpreted as a 24-bit quantity and placed in the three rightmost bytes of the network address. This makes the two-part address format convenient for specifying Class A network addresses as net.host.

When a one-part address is specified, the value is stored directly in the network address space without any rearrangement of its bytes.

Numbers supplied as address parts in standard dotted-decimal notation can be decimal, hexadecimal, or octal. Numbers are interpreted in C language syntax. A leading 0x implies hexadecimal; a leading 0 implies octal. A number without a leading 0 implies decimal.

Applications that use the inet_addr() call can enter field values exceeding the above restrictions. The call accepts the least significant bits up to an integer in length, then checks whether the truncated value exceeds the maximum field value. For example, if an application enters a field value of 0x1234567890 and the system uses 16 bits per integer, then the inet_addr() call uses bits 0-15. The call returns 0x34567890.

Applications can omit field values between delimiters. The inet_addr() call interprets empty fields as 0.

Notes:

1. The inet_addr() call does not check the pointer to the ASCII string. The user must ensure the validity of the address in the ASCII string.

2. The application must verify that the network and host IDs for the internet address conform to either a Class A, B, or C internet address. The inet_attr() call processes any other class of address as a Class C address.

Return Values

The internet address is returned in network-byte order. The return value points to static data that subsequent API calls can modify.

For valid input strings, the inet_addr() call returns an unsigned long value comprised of the bit patterns of the input fields concatenated together. The call places the first pattern in the most significant position and appends any subsequent patterns to the next most significant positions.

The inet_addr() call returns an error value of -1 for invalid strings.

Note: An internet address with a dotted-decimal notation value of 255.255.255.255 or its equivalent in a different base format causes the inet_addr() call to return an unsigned long value of 4294967295. This value is identical to the unsigned representation of the error value. Otherwise, the inet_addr() call considers 255.255.255.255 a valid internet address.

Related Calls

endhostent()
endnetent()
gethostbyaddr()
gethostbyname()
getnetbyaddr()
getnetbyname()
getnetent()
inet_lnaof()
inet_makeaddr()
inet_netof()
inet_network()
inet_ntoa()
sethostent()
setnetent()

----

inet_lnaof()

The inet_lnaof() call returns the local network portion of an internet host address.

Syntax

```
#include <types.h>
#include <arpa\inet.h>
u_long inet_lnaof(in)
struct in_addr in;
```

Parameters

in
Host internet address.

Description

The inet_lnaof() call masks off the host ID of an internet address based on the internet address class. The calling application must enter the internet address as an unsigned long value.

Note: The application must verify that the network and host IDs for the internet address conform to either a Class A, B, or C internet address. The inet_lnaof() call processes any other class of address as a Class C address.

Return Values

The local network address is returned in host-byte order. The return value points to static data that subsequent API calls can modify.

The return values of the inet_lnaof() call depend on the class of internet address the application provides:

Class A	The logical product of the internet address and 0x00FFFFFF
Class B	The logical product of the internet address and 0x0000FFFF
Class C	The logical product of the internet address and 0x000000FF

Related Calls

endhostent()
endnetent()
gethostbyaddr()
gethostbyname()
getnetbyaddr()
getnetbyname()
getnetent()
inet_addr()
inet_makeaddr()
inet_netof()
inet_network()
inet_ntoa()
sethostent()
setnetent()

----

inet_makeaddr()

The inet_makeaddr() call constructs an internet address from a network number and a local network address.

Syntax

```
#include <types.h>
#include <arpa\inet.h>
struct in_addr inet_makeaddr(net, lna)
u_long net;
u_long lna;
```

Parameters

net
: Network number.

lna
: Local network address.

Description

The inet_makeaddr() call forms an internet address from the network ID and host ID provided by the application (as integer types). If the application provides a Class A network ID, the inet_makeaddr() call forms the internet address using the net ID in the highest-order byte, and the logical product of the host ID and 0x00FFFFFF in the 3 lowest-order bytes. If the application provides a Class B network ID, the inet_makeaddr() call forms the internet address using the net ID in the two highest-order bytes, and the logical product of the host ID and 0x0000FFFF in the lowest two ordered bytes. If the application does not provide either a Class A or Class B network ID, the inet_makeaddr() call forms the internet address using the network ID in the 3 highest-order bytes, and the logical product of the host ID and 0x0000FFFF in the lowest-ordered byte.

The inet_makeaddr() call ensures that the internet address format conforms to network order, with the first byte representing the high-order byte. The inet_makeaddr() call stores the internet address in the structure as an unsigned long value.

The application must verify that the network ID and host ID for the internet address conform to class A, B, or C. The inet_makeaddr() call processes any other class of address as a Class C address.

The inet_makeaddr() call expects the in_addr structure to contain only the internet address field. If the application defines the in_addr structure otherwise, then the value returned in in_addr by the inet_makeaddr() call is undefined.

Return Values

The internet address is returned in network-byte order. The return value points to static data that subsequent API calls can modify.

If the inet_makeaddr() call is unsuccessful, the call returns a value of -1.

Related Calls

endhostent()
endnetent()
gethostbyaddr()
gethostbyname()
getnetbyaddr()
getnetbyname()
getnetent()
inet_addr()
inet_lnaof()
inet_netof()
inet_network()
inet_ntoa()
sethostent()
setnetent()

----

inet_netof()

The inet_netof() call returns the network number portion of the internet host address in network-byte order.

Syntax

```
#include <types.h>
```

```
#include <arpa\inet.h>
u_long inet_netof(in)
struct in_addr in;
```

Parameters

in
: Internet address in network-byte order.

Description

The inet_netof() call returns the network number from the specified internet address number, typed as unsigned long value. The inet_netof() call masks off the network number and the host number from the internet address, based on the internet address class.

Note: The application assumes responsibility for verifying that the network number and the host number for the internet address conforms to a class A or B or C internet address. The inet_netof() call processes any other class of address as a class C address.

Return Values

The network number is returned in host-byte order. The return value points to static data that subsequent API calls can modify.

When successful, the inet_netof() call returns a network number from the specified long value representing the internet address. If the application gives a class A internet address, the inet_lnoaf() call returns the logical product of the internet address and 0xFF000000. If the application gives a class B internet address, the inet_lnoaf() call returns the logical product of the internet address and 0xFFFF0000. If the application does not give a class A or B internet address, the inet_lnoaf() call returns the logical product of the internet address and 0xFFFFFF00.

Related Calls

endhostent()
endnetent()
gethostbyaddr()
gethostbyname()
getnetbyaddr()
getnetbyname()
getnetent()
inet_addr()
inet_lnaof()
inet_makeaddr()
inet_network()
inet_ntoa()
sethostent()
setnetent()

----

inet_network()

The inet_network() call constructs a network number from character strings representing numbers expressed in standard dotted-decimal notation.

Syntax

```
#include <types.h>
#include <arpa\inet.h>
u_long inet_network(cp)
char *cp;
```

Parameters

cp
: A character string in standard dotted-decimal notation.

Description

The inet_network() call converts an ASCII string containing a valid internet address using dotted-decimal notation (such as 120.121.122.123) to an internet address number formatted as an unsigned long value. The inet_network() call returns an error value if the application does not provide an ASCII string containing a valid internet address using dotted-decimal notation.

The input ASCII string must represent a valid internet address number, as described in Internet Address Formats. The input string must be terminated with a null terminator (0x00) or a space (0x30). The inet_network() call ignores characters that follow the terminating character.

The input string can express an internet address number in decimal, hexadecimal, or octal format. In hexadecimal format, the string must begin with 0x. The string must begin with 0 to indicate octal format. In decimal format, the string requires no prefix.

Each octet of the input string must be delimited from another by a period. The application can omit values between delimiters. The inet_network() call interprets missing values as 0.

The following examples show valid strings and their output values in both decimal and hexadecimal notation:

Examples of Valid Strings

Input string	Output value (in decimal)	Output value (in hex)
...1	1	0x00000001
.1..	65536	0x00010000
1	256	0x00000100
0xFFFFFFFF	255	0x000000FF
1.	16777216	0x01000000
1.2.3.4	16909060	0x01020304
0x01.0X2.03.004	16909060	0x01020304
1.2. 3.4	16777218	0x01000002
9999.1.1.1	251724033	0x0F010101

The following examples show invalid input strings and the reasons they are not valid:

Examples of Invalid Strings

Input string	Reason string is not valid
1.2.3.4.5	Excessive fields.
1.2.3.4.	Excessive delimiters (and therefore fields).
1,2	Bad delimiter.
1p	String not terminated by null terminator nor space.
{empty string}	No field or delimiter present.

Typically, the value of each octet of an internet address cannot exceed 246. The inet_network() call can accept larger values, but it uses only the eight least significant bits for each field value. For example, if an application passes 0x1234567890.0xabcdef, the inet_network() call returns 37103 (0x000090EF).

The application must verify that the network ID and host ID for the internet address conform to class A, class B, or class C. The inet_makeaddr() call processes any nonconforming class of address as a class C address.

The inet_network() call does not check the pointer to the ASCII input string. The application must verify the validity of the address of the string.

Return Values

The network number is returned in host-byte order. The return value points to static data that subsequent API calls can modify.

For valid input strings, the inet_network() call returns an unsigned long value that comprises the bit patterns of the input fields concatenated together. The inet_network() call places the first pattern in the leftmost (most significant) position and appends subsequent patterns if they exist.

For invalid input strings, the inet_network() call returns a value of -1.

Related Calls

endhostent()
endnetent()
gethostbyaddr()
gethostbyname()
getnetbyaddr()
getnetbyname()
getnetent()
inet_addr()
inet_netof()
inet_makeaddr()
inet_netof()
inet_ntoa()
sethostent()
setnetent()

----

inet_ntoa()

The inet_ntoa() call returns a pointer to a string in dotted-decimal notation.

Syntax

```
#include <types.h>
#include <arpa\inet.h>
char *inet_ntoa(in)
struct in_addr in;
```

Parameters

in
Host internet address.

Description

This call returns a pointer to a string expressed in the dotted-decimal notation. The inet_ntoa() call accepts an internet address expressed as a 32-bit quantity in network-byte order and returns a string expressed in dotted-decimal notation. All internet addresses are returned in network order, with the first byte being the high-order byte.

Use C-language integers when specifying each part of a dotted-decimal notation.

Return Values

Returns a pointer to the internet address expressed in dotted-decimal notation. The return value points to static data that subsequent API calls can modify.

If the inet_ntoa() call is unsuccessful, the call returns a value of -1.

Related Calls

endhostent()
endnetent()
gethostbyaddr()
gethostbyname()
getnetbyaddr()
getnetbyname()
getnetent()
inet_addr()
inet_netof()

inet_makeaddr()
inet_network()
sethostent()
setnetent()

----

ntohl()

The ntohl() call translates a long integer from network-byte order to host-byte order.

Syntax

```
#include <types.h>
u_long ntohl(a)
u_long a;
```

Parameters

a
Unsigned long integer to be put into host-byte order.

Description

This call translates a long integer from network-byte order to host-byte order. Receiving hosts require addresses and ports in host-byte order. Use the ntohl() call to convert internet addresses to the host integer representation.

Return Values

Returns the translated long integer.

Related Calls

endhostent()
endservent()
gethostbyaddr()
gethostbyname()
getservbyname()
getservbyport()
getservent()
htonl()
htons()
ntohs()
sethostent()
setservent()

----

ntohs()

The ntohs() call translates a short integer from network-byte order to host-byte order.

Syntax

```
#include <types.h>
u_short ntohs(a)
u_short a;
```

Parameters

a
: Unsigned short integer to be put into host-byte order.

Description

This call translates a short integer from network-byte order to host-byte order. Receiving hosts require addresses and ports in host-byte order. Use the ntohs() call to convert ports to the host integer representation.

Return Values

The ntohs() call returns the translated short integer.

Related Calls

- endhostent()
- endservent()
- gethostbyaddr()
- gethostbyname()
- getservbyname()
- getservbyport()
- getservent()
- htonl()
- htons()
- ntohl()
- sethostent()
- setservent()

---

putlong()

The putlong() call places long byte quantities into the byte stream.

Syntax

```
#include <sys\types.h>
#include <netinet\in.h>
#include <arpa\nameser.h>
#include <resolv.h>
void putlong(l, msgp)
u_long l;
u_char *msgp;
```

Parameters

l
: Represents a 32-bit integer.

msgp
: Represents a pointer into the byte stream.

Description

The putlong() call places long byte quantities into the byte stream or arbitrary byte boundaries.

The putlong() call is one of a group of calls that form the resolver, a set of functions that resolve domain names. Global information used by the resolver calls is kept in the _res data structure. The <RESOLV.H> file contains the _res structure definition. (See The _res Data Structure for more information.)

Related Calls

- dn_comp()
- dn_expand()
- dn_find()
- dn_skipname()
- _getlong()
- _getshort()

putshort()
res_init()
res_mkquery()
res_query()
res_search()
res_send()

----

putshort()

The putshort() call places short byte quantities into the byte stream.

Syntax

```
#include <sys\types.h>
#include <netinet\in.h>
#include <arpa\nameser.h>
#include <resolv.h>
void putshort (s, msgp)
u_short s;
u_char *msgp;
```

Parameters

s
Represents a 16-bit integer.

msgp
Represents a pointer into the byte stream.

Description

The putshort() call puts short byte quantities into the byte stream or arbitrary byte boundaries.

The putshort() call is one of a group of calls that form the resolver, a set of functions that resolve domain names. Global information used by the resolver calls is kept in the _res data structure. The <RESOLV.H> file contains the _res structure definition. (See The _res Data Structure for more on the _res data structure.)

Related Calls

dn_comp()
dn_expand()
dn_find()
dn_skipname()
_getlong()
_getshort()
putlong()
res_init()
res_mkquery()
res_query()
res_send()

Raccept()

The Raccept() socket call accepts a connection request from a SOCKS server.

Syntax

```
#include <types.h>
```

```
#include <sys\socket.h>
#include <netinet\in.h>
int Raccept(s, name, namelen)
int s;
sockaddr *name;
int *namelen;
```

Parameters

s
Socket descriptor.

name
Pointer to a sockaddr structure that contains the socket address of the connection client when the Raccept() call returns. The format of *name* is determined by the communications domain where the client resides. This parameter can be NULL if the caller is not interested in the client address.

namelen
Must initially point to an integer that contains the size in bytes of the storage pointed to by *name*. On return, that integer contains the size of the data returned in the storage pointed to by *name*. If *name* is NULL, *namelen* is ignored and can be NULL.

Description

This call is used to accept the first pending connection on the socket. If the socket was bound through a SOCKS server using Rbind(), it will block until the SOCKS server accepts a connection on its behalf. If the socket was not bound through a SOCKS server, this call is equivalent to accept().

Return Values

A non-negative socket descriptor indicates success; the value -1 indicates an error. You can get the specific error code by calling sock_errno() or psock_errno().

Error Code	**Description**
SOCENOTSOCK	The *s* parameter is not a valid socket descriptor.
SOCEFAULT	Using *name* and *namelen* would result in an attempt to copy the address into a portion of the caller address space into which information cannot be written.
SOCEINTR	Interrupted system call.
SOCEINVAL	Listen() was not called for socket *s*.
SOCENOBUFS	Insufficient buffer space available to create the new socket.
SOCEOPNOTSUPP	The *s* parameter is not connection-oriented.
SOCEWOULDBLOCK	The *s* parameter is in nonblocking mode and no connections are on the queue.
SOCECONNABORTED	The software caused a connection close.

Related Calls

accept()
connect()
Rbind()
Rconnect()
Rgetsockname()
Rlisten()
sock_errno()
socket()

Rbind()

The Rbind() call binds a local name to the socket.

Syntax

```
#include <types.h>
#include <sys\socket.h>
int Rbind(s, name, namelen remoteaddr)
int s;
struct sockaddr *name;
int namelen;
struct sockaddr *remoteaddr;
```

Parameters

s
: Socket descriptor returned by a previous call to socket().

name
: Pointer to a *sockaddr* structure containing the name that is to be bound to *s*.

namelen
: Size in bytes of the *sockaddr* structure pointed to by *name*.

remoteaddr
: Pointer to a *sockaddr* structure containing the address of the remote host from which the connection is expected to be established.

Description

This call checks *remoteaddr* to see if it is reachable via a direct connection, or whether a connection through a SOCKS server is required. If the former, then this call is equivalent to a `bind (s, name, namelen)`. If the latter, then a connection to the SOCKS server is established and a bind request is sent. Rgetsockname() may be called to retrieve the IP address and port number the SOCKS server assigned.

Related Calls

bind()
htons()
inet_addr()
Rconnect()
Rgethostbyname()
Rgetsockname()
Rlisten()
sock_errno()
socket()

Rconnect()

The Rconnect() socket call requests a connection to a remote host. The request goes directly to a SOCKS server.

Syntax

```
#include <types.h>
#include <sys\socket.h>
int Rconnect(s, name, namelen)
```

Description

See the "Description" section in connect() for additional information about connecting. (This description information also applies to the Rconnect() call).

Related Calls

connect()
Raccept()
Rbind()

Rgetsockname()
send()
socket()

----

res_init()

The res_init() call initializes the default domain name.

Syntax

```
#include <types.h>
#include <netinet\in.h>
#include <arpa\nameser.h>
#include <resolv.h>
void res_init()
```

Description

This call reads the ETC\RESOLV and ETC\RESOLV2 files for the domain name information and for the IP addresses of the initial hosts running the name server. If none of these files exist, name resolution will use the ETC\HOSTS file. One of these files should be operational.

Note: If the ETC\RESOLV files do not exist, the res_init() call attempts name resolution using the local ETC\HOSTS file. If the system is not using a domain name server, the ETC\RESOLV file should not exist. The ETC\HOSTS file should be present on the system even if the system is using a name server. In this instance, the file should contain the host IDs that the system requires to function even if the name server is not functioning.

The res_init() call is one of a group of calls that form the resolver. The resolver is a set of functions that perform a translation between domain names and network addresses. Global information used by the resolver calls resides in the _res data structure. The <RESOLV.H> file contains the _res data structure definition. (See The _res Data Structure for more on the _res data structure.)

Related Calls

dn_comp()
dn_expand()
dn_find()
dn_skipname()
_getlong()
_getshort()
putlong()
putshort()
res_mkquery()
res_query()
res_search()
res_send()

----

res_mkquery()

The res_mkquery() call makes a query message for the name servers in the internet domain.

Syntax

```
#include <types.h>
#include <netinet\in.h>
#include <arpa\nameser.h>
#include <resolv.h>
int res_mkquery(op, dname, class, type, data, datalen, newrr,
            buf, buflen)
```

```
int op;
char *dname;
int class;
int type;
char *data;
int datalen;
struct rrec *newrr;
char *buf;
int buflen;
```

Parameters

op

The usual type is QUERY, but you can set the parameter to any query type defined in the <ARPA\NAMESER.H> header file.

dname

Pointer to the domain name. If *dname* points to a single label and the RES_DEFNAMES bit in the _res data structure (see The _res Data Structure) defined in the <RESOLV.H> header file is set, the call appends *dname* to the current domain name. The current domain name is defined in the ETC\RESOLV file.

class

One of the following values:

C_IN	ARPA Internet
C_CHAOS	Chaos network at MIT

type

One of the following type values for resources and queries:

T_A	Host address
T_NS	Authoritative server
T_MD	Mail destination
T_MF	Mail forwarder
T_CNAME	Canonical name
T_SOA	Start of authority zone
T_MB	Mailbox domain name
T_MG	Mail group member
T_MR	Mail rename name
T_NULL	Null resource record
T_WKS	Well-known service
T_PTR	Domain name pointer
T_HINFO	Host information
T_MINFO	Mailbox information
T_MX	Mail routing information
T_UINFO	User information
T_UID	User ID
T_GID	Group ID

data

Pointer to the data sent to the name server as a search key.

datalen

Size of the *data* parameter in bytes.

newrr

Reserved. Unused parameter.

buf

Pointer to the query message.

buflen

Length of the buffer in bytes pointed to by the *buf* parameter.

Description

This call creates packets for name servers in the internet domain. The call makes a query message for the name servers and puts that query message in the location pointed to by the *buf* parameter.

The res_mkquery() call is one of a group of calls that form the resolver. The resolver is a set of functions that perform a translation between domain names and network addresses. Global information used by the resolver calls resides in the _res data structure. The <RESOLV.H> file contains the _res data structure definition. (See The _res Data Structure for more on the _res data structure.)

Return Values

When successful, the res_mkquery() call returns the size of the query. When the query is larger than the value of *buflen* , the call fails and returns a value of -1.

Related Calls

- dn_comp()
- dn_expand()
- dn_find()
- dn_skipname()
- _getlong()
- _getshort()
- putlong()
- putshort()
- res_init()
- res_query()
- res_search()
- res_send()

----

res_query()

The res_query() call provides an interface to the server query mechanism.

Syntax

```
#include <sys\types.h>
#include <netinet\in.h>
#include <arpa\nameser.h>
#include <resolv.h>
int res_query(name, class, type, answer, anslen)
char *name;
int class;
int type;
u_char *answer;
int anslen;
```

Parameters

name
Points to the name of the domain. If the *name* parameter points to a single-component name and the RES_DEFNAMES structure is set, as it is by default, the call appends the default domain name to the single-component name. The current domain name is defined by the name server in use or is specified in the ETC\RESOLV file.

class
Specifies one of the following values:

C_IN	Specifies the ARPA Internet
C_CHAOS	Specifies the Chaos network at MIT

type
Requires one of the following values:

T_A	Host address
T_NS	Authoritative server
T_MD	Mail destination
T_MF	Mail forwarder
T_CNAME	Canonical name
T_SOA	Start-of-authority zone
T_MB	Mailbox-domain name
T_MG	Mail-group member
T_MR	Mail-rename name
T_NULL	Null resource record
T_WKS	Well-known service
T_PTR	Domain name pointer

T_HINFO	Host information
T_MINFO	Mailbox information
T_MX	Mail-routing information
T_UINFO	User (finger command) information
T_UID	User ID
T_GID	Group ID

answer
Points to an address where the response is stored.

anslen
Specifies the size of the answer buffer.

Description

The res_query() call provides an interface to the server query mechanism. It constructs a query, sends it to the local server, awaits a response, and makes preliminary checks on the reply. The query requests information of the specified type and class for the fully-qualified domain name specified in the *name* parameter. The reply message is left in the answer buffer whose size is specified by the *anslen* parameter, which is supplied by the caller.

The caller must parse *answer* and determine whether it answers the question.

The res_query() call is one of a group of calls that form the resolver, a set of functions that resolve domain names. The _res data structure contains global information used by the resolver calls. The <RESOLV.H> file contains the _res structure definition. (See The _res Data Structure for more on the _res data structure.)

Return Values

When successful, the res_query() call returns the size of the response. When unsuccessful, the res_query() call returns a value of -1 and sets the h_errno value to the appropriate error.

Related Calls

- dn_comp()
- dn_expand()
- dn_find()
- dn_skipname()
- _getlong()
- _getshort()
- putlong()
- putshort()
- res_init()
- res_mkquery()
- res_search()
- res_send()

----

res_querydomain()

The res_querydomain() call queries the concatenation of *name* and *domain* .

Syntax

```
#include <sys\types.h>
#include <netinet\in.h>
#include <arpa\nameser.h>
#include <resolv.h>
int res_querydomain(name, domain class, type, answer, anslen)
char *name;
char *domain;
int class;
int type;
u_char *answer;
int anslen;
```

Parameters

domain
Domain name.

name
Host name.

class
Specifies one of the following values:

C_IN	Specifies the ARPA Internet.
C_CHAOS	Specifies the Chaos network at MIT.

type
Requires one of the following values:

T_A	Host address
T_NS	Authoritative server
T_MD	Mail destination
T_MF	Mail forwarder
T_CNAME	Canonical name
T_SOA	Start-of-authority zone
T_MB	Mailbox-domain name
T_MG	Mail-group member
T_MR	Mail-rename name
T_NULL	Null resource record
T_WKS	Well-known service
T_PTR	Domain name pointer
T_HINFO	Host information
T_MINFO	Mailbox information
T_MX	Mail-routing information
T_UINFO	User (finger command) information
T_UID	User ID
T_GID	Group ID

answer
Points to an address where the response is stored.

anslen
Specifies the size of the answer buffer.

Description

The res_querydomain() call concatenates *name* and *domain,* removing a trailing dot from *name* if *domain* is null. The res_querydomain() call then calls res_query() to build and perform the query.

Return Values

The value 0 indicates success; the value -1 indicates an error. The h_errno() value is set to the appropriate error code.

Related Calls

dn_comp()
dn_expand()
dn_find()
dn_skipname()
_getlong()
_getshort()
putlong()
putshort()
res_init()
res_mkquery()
res_search()
res_send()

----

res_search()

The res_search() call makes a query and awaits a response.

Syntax

```
#include <sys\types.h>
#include <netinet\in.h>
#include <arpa\nameser.h>
#include <resolv.h>
int res_search(name, class, type, answer, anslen)
char *name;
int class;
int type;
u_char *answer;
int anslen;
```

Parameters

name
Points to the name of the domain. If the *name* parameter points to a single-component name and the RES_DEFNAMES structure is set, as it is by default, the call appends the default domain name to the single-component name. The current domain name is defined by the name server in use or is specified in the ETC\RESOLV file.

If the RES_DNSRCH bit is set, as it is by default, the res_search() call searches for host names in both the current domain and in parent domains.

class
Specifies one of the following values:

C_IN	Specifies the ARPA Internet.
C_CHAOS	Specifies the Chaos network at MIT.

type
Requires one of the following values:

T_A	Host address
T_NS	Authoritative server
T_MD	Mail destination
T_MF	Mail forwarder
T_CNAME	Canonical name
T_SOA	Start-of-authority zone
T_MB	Mailbox-domain name
T_MG	Mail-group member
T_MR	Mail-rename name
T_NULL	Null resource record
T_WKS	Well-known service
T_PTR	Domain name pointer
T_HINFO	Host information
T_MINFO	Mailbox information
T_MX	Mail-routing information
T_UINFO	User (finger command) information
T_UID	User ID
T_GID	Group ID

answer
Points to an address where the response is stored.

anslen
Specifies the size of the answer buffer.

Description

The res_search() call makes a query and awaits a response, like the res_query() call. However, it also implements the default and search rules controlled by the RES_DEFNAMES and RES_DNSRCH options.

Note: This call is only useful for queries in the same name hierarchy as the local host.

The res_search() call is one of a group of calls that form the resolver, a set of functions that resolve domain names. The _res data structure contains global information used by the resolver calls. The <RESOLV.H> file contains the _res structure definition. (See The _res Data Structure for more on the _res data structure.)

Return Values

When successful, the res_search() call returns the size of the response. When unsuccessful, the res_search() call returns a value of -1 and sets the h_errno value to the appropriate error.

Related Calls

- dn_comp()
- dn_expand()
- dn_find()
- dn_skipname()
- _getlong()
- _getshort()
- putlong()
- putshort()
- res_init()
- res_mkquery()
- res_query()
- res_send()

----

res_send()

The res_send() call sends a query to a local name server.

Syntax

```
#include <types.h>
#include <netinet\in.h>
#include <arpa\nameser.h>
#include <resolv.h>
int res_send(msg, msglen, ans, anslen)
char *msg;
int msglen;
char *ans;
int anslen;
```

Parameters

msg
Pointer to the beginning of a message.

msglen
Length of the buffer in bytes pointed to by the *msg* parameter.

ans
Pointer to the location where the received response is stored.

anslen
Length of the buffer in bytes pointed by the *ans* parameter.

Description

This call sends a query to the local name server and calls the res_init() call if the RES_INIT option of the global _res structure is not set. It also handles timeouts and retries.

The res_send() call is one of a set of calls that form the resolver. The resolver is a group of functions that perform a translation between domain names and network addresses. Global information used by the resolver calls resides in the _res data structure. The <RESOLV.H> file contains the _res data structure definition. (See The _res Data Structure for more on the _res data structure.)

Return Values

When successful, the call returns the length of the message. When unsuccessful, the call returns a value of -1.

Related Calls

dn_comp()
dn_expand()
dn_find()
dn_skipname()
_getlong()
_getshort()
putlong()
putshort()
res_init()
res_mkquery()
res_query()
res_search()

----

rexec()

The rexec() call allows command processing on a remote host.

Syntax

```
#include <utils.h>
int rexec(host, port, user, passwd, cmd, err_sd2p)
char **host;
int port;
char *user, *passwd, *cmd;
int *err_sd2p;
```

Parameters

host
Contains the name of a remote host that is listed in the ETC\HOSTS file or ETC\RESOLV file. If the name of the host is not found in either file, the rexec() call is unsuccessful.

port
Specifies the well-known internet port to use for the connection. A pointer to the structure that contains the necessary port can be obtained by issuing the following library call:

```
getservbyname("exec","tcp")
```

When directly specifying an integer for the port, specify it in Network Byte order, obtainable by using the htons() function call.

user
Points to a user ID valid at the remote host.

passwd
Points to a password valid at the remote host.

cmd
Points to the name of the command to be processed at the remote host.

err_sd2p
Points to an error socket descriptor. An auxiliary channel to a control process is set up, and a descriptor for it is placed in the *err_sd2p* parameter. The control process provides diagnostic output from the remote command on this channel. This diagnostic information does not include remote authorization failure, since this connection is set up after authorization has been verified.

Description

The rexec() call allows the calling process to start commands on a remote host.

If the rexec() connection succeeds, a socket in the internet domain of type SOCK_STREAM is returned to the calling process and is given to the remote command as standard input and standard output.

Return Values

When successful, the call returns a socket to the remote command.

When unsuccessful, the call returns a value of -1, indicating that the specified host name does not exist.

Examples

```
int normsock;
char *host = NULL, *luser = NULL, *password = NULL, *cmd;
struct servent *sp;
sp = getservbyname("exec", "tcp");

host = "remote _host";
luser = "my _userid";
password = "my _password";
cmd = "rempte_host_cmd";

normsock = rexec(&host, sp->s_port, luser,password, cmd, &errsock);
if (normsock == -1)
       exit(-1);
```

Related Calls

getservbyname()

Rgethostbyname()

The Rgethostbyname() call returns a pointer to information about a host specified by a host name. The request goes directly to a SOCKS server, **sockd.**

Syntax

```
#include <netdb.h>
struct hostent *Rgethostbyname(name)
char *name;
```

See the "Description" section in gethostbyname() for additional information about this topic. (This description information also applies to the Rgethostbyname() call.)

Related Calls

endhostent()
gethostbyaddr()
gethostent()
inet_addr()
sethostent()

Rgetsockname()

Rgetsockname() gets the socket name from the SOCKS server.

Syntax

```
#include <types.h>
#include <sys\socket.h>
int Rgetsockname(s, name, namelen)
int s;
```

```
struct sockaddr *name;
int *namelen;
```

Parameters

s
: Socket descriptor.

name
: Pointer to a sockaddr structure. The name of *s* is returned.

namelen
: Pointer to the size in bytes of the buffer pointed to by *name* .

Description

This call returns the name the SOCKS server bound to the socket after a successful call to Rbind() or connect(). If the socket is not connected through a SOCKS server, this call is equivalent to getsockname().

Return Values

The value 0 indicates success; the value -1 indicates an error. You can get the specific error code by calling sock_errno() or psock_errno().

Error Code	**Description**
SOCENOTSOCK	The *s* parameter is not a valid socket descriptor.
SOCEFAULT	Using the *name* and *namelen* parameters as specified would result in an attempt to access storage outside of the address space of the caller.
SOCENOBUFS	No buffer space available.

Related Calls

getpeername()
getsockname()
Raccept()
Rbind()
Rconnect()
sock_errno()
socket()

Rlisten()

The Rlisten() socket call completes the binding necessary for a socket to accept connections and creates a connection request queue for incoming requests. This call goes directly to the SOCKS server, rather than querying the SOCKS flag (socks_flag).

Syntax

```
#include <types.h>
#include <sys\socket.h>
#include <netinet\in.h>
int Rlisten(s, backlog)
int s;
int backlog;
```

Parameters

s
: Socket descriptor.

backlog
: Controls the maximum queue length for pending connections.

Description

This call checks to see if the socket is connected through a SOCKS server, and if not, calls listen(). If it is, the call has no effect.

Return Values

The value 0 indicates success; the value -1 indicates an error. You can get the specific error code by calling sock_errno() or psock_errno().

Error Code	**Description**
SOCENOTSOCK	The *s* parameter is not a valid socket descriptor.
SOCEOPNOTSUPP	The *s* parameter is not a socket descriptor that supports the listen() call.

Related Calls

- listen()
- Raccept()
- Rbind()
- Rconnect()
- sock_errno()
- socket()

----

sethostent()

The sethostent() call opens and rewinds the ETC\HOSTS file.

Syntax

```
#include <netdb.h>
void sethostent(stayopen)
int stayopen;
```

Parameters

stayopen
Allows the ETC\HOSTS file to stay open after each call; specifying 0 closes the file.

Description

This call opens and rewinds the ETC\HOSTS file. If the *stayopen* parameter is nonzero, the ETC\HOSTS file stays open after each of the gethost calls.

Related Calls

- endhostent()
- gethostbyaddr()
- gethostbyname()
- gethostent()

----

setnetent()

The setnetent() call opens and rewinds the ETC\NETWORKS file.

Syntax

```
#include <netdb.h>
void setnetent(stayopen)
int stayopen;
```

Parameters

stayopen
Allows the ETC\NETWORKS file to stay open after each call; specifying 0 closes the file.

Description

This call opens and rewinds the ETC\NETWORKS file, which contains information about known networks. If the *stayopen* parameter is nonzero, the ETC\NETWORKS file stays open after each of the getnet calls.

Related Calls

- endnetent()
- getnetbyaddr()
- getnetbyname()
- getnetent()

----

setprotoent()

The setprotoent() call opens and rewinds the ETC\PROTOCOL file.

Syntax

```
#include <netdb.h>
void setprotoent(stayopen)
int stayopen;
```

Parameters

stayopen
Allows the ETC\PROTOCOL file to stay open after each call; specifying 0 closes the file.

Description

This call opens and rewinds the ETC\PROTOCOL file, which contains information about known protocols. If the *stayopen* parameter is nonzero, the ETC\PROTOCOL file stays open after each of the getproto calls.

Related Calls

- endprotoent()
- getprotobyname()
- getprotobynumber()
- getprotoent()

----

setservent()

The setservent() call opens and rewinds the ETC\SERVICES file.

Syntax

```
#include <netdb.h>
void setservent(stayopen)
int stayopen;
```

Parameters

stayopen
Allows the ETC\SERVICES file to stay open after each call; specifying 0 closes the file.

Description

This call opens and rewinds the ETC\SERVICES file, which contains information about known services and well-known ports. If the *stayopen* parameter is nonzero, the ETC\SERVICES file stays open after each of the getserv calls.

Related Calls

endprotoent()
endservent()
getprotobyname()
getprotobynumber()
getprotoent()
getservbyname()
getservbyport()
getservent()
setprotoent()

R0LIB32 Library

The following include files are available:

- r0lib.h
- nerrno.h

The following socket calls are available:

- **long sock_init(void);**
- **long socket(long, long,long);**
- **long bind(long,char *, long);**
- **long connect(long, char *, long);**
- **long listen(long,long);**
- **long accept(long, char *, long *);**
- **long sendto(long, char *, long, long, char *, long);**
- **long send(long, char *, long, long);**
- **long sendmsg(long, char *, long);**
- **long recvfrom(long, char *, long, long, char *, long *);**
- **long recv(long, char *, long, long);**
- **long recvmsg(long, char *, long);**
- **long shutdown(long, long);**
- **long setsockopt(long, long, long, char *, long);**
- **long getsockopt(long, long, long, char *, long *);**
- **long getsockname(long, char *, long *);**
- **long getpeername(long, char *, long *);**

- **ulong gethostid(void);**
- **long soclose(long);**
- **long ioctl(long, long, char *, long);**
- **long select(long *, long, long, long, long);**

sock_init() returns 0 for success else failure. **sock_init()** must be called before any other socket call is made. Internally **sock_init()** attaches to INET$ and prepares the library to do socket calls. If **sock_init()** fails, it means that INET$ (sockets.sys) is not loaded. Check your config.sys "device=" statement, or execute the "inetver" command at OS/2 command prompt. The command should return a TCP/IP version greater than 3.00. Link your Ring-0 code with r0lib32.lib and you will be ready to use all sockets API calls.

Example

To test **r0lib32.lib** build test.sys, add an entry for the device driver in config.sys, and copy test.sys in \mptn\protocol directory. Reboot the machine so that test.sys is loaded in memory. Now compile testini.c and generate testini.exe.

By running testini.exe the function test_server(), in driver.c is invoked, which acts as a server and waits for the client at port no 5000. Compile and run the client program (client.c) on any other machine.

The test_server opens a STREAM socket in the internet domain and waits for a client at port no 5000. In the client program, the IP address of the server machine that has test.sys is to be specified while filling the structure sockaddr_in. To compile the client program use makefile.cli.

For more information:

The **TCP/IP Toolkit** contains a programming example in detail, which exploits the use of **r0lib32.**

Remote Procedure and eXternal Data Representation API

The following table briefly describes each remote procedure and XDR call, and identifies where you can find the syntax, parameters, and other appropriate information for these calls.

Remote Procedure and XDR API Quick Reference

RPC or XDR Call	Description
auth_destroy()	Destroys authentication information
authnone_create()	Creates and returns a NULL RPC authentication handle
authunix_create()	Creates and returns a UNIX-based authentication handle
authunix_create_default()	Calls authunix_create() with default parameters
callrpc()	Calls remote procedures
clnt_broadcast()	Broadcasts a remote program to all locally connected broadcast networks
clnt_call()	Calls the remote procedure (*procnum*) associated with the client handle (*clnt*)
clnt_destroy()	Destroys a client's RPC handle
clnt_freeres()	Deallocates resources assigned for decoding the results of an RPC
clnt_geterr()	Copies the error structure from a client's handle to the local structure

clnt_pcreateerror()	Indicates why a client handle cannot be created
clnt_perrno()	Writes a message to the standard error device corresponding to the condition indicated by *stat*
clnt_perror()	Writes an error message indicating why RPC failed
clntraw_create()	Creates a client transport handle to use in a single task
clnttcp_create()	Creates an RPC client transport handle for the remote program using TCP transport
clntudp_create()	Creates an RPC client transport handle for the remote program using UDP transport
get_myaddress()	Returns the local host's internet address
getrpcbyname()	Returns an RPC program entry specified by a name in the RPC file
getrpcbynumber()	Returns an RPC program entry specified by a number in the RPC file
getrpcent()	Returns the next entry in the TCPIP\ETC\RPC file
pmap_getmaps()	Returns a list of current program-to-port mappings on a specified remote host's Portmapper
pmap_getport()	Returns the port number associated with the remote program, the version, and the transport protocol
pmap_rmtcall()	Instructs Portmapper to make an RPC call to a procedure on a host on your behalf
pmap_set()	Sets the mapping of the program to *port* on the local machine's Portmapper
pmap_unset()	Removes the mappings associated with *prognum* and *versnum* on the local machine's Portmapper
registerrpc()	Registers a procedure with the local Portmapper and creates a control structure to remember the server procedure and its XDR routine
rpc_createerr	A global variable that is set when any RPC client creation routine fails
svc_destroy()	Deletes the RPC service transport handle *xprt*, which becomes undefined after this routine is called
svc_freeargs()	Frees storage allocated to decode the arguments received by svc_getargs()
svc_getargs()	Uses the XDR routine *inproc* to decode the arguments of an RPC

	request associated with the RPC service transport handle *xprt*
svc_getcaller()	Gets the network address of the client associated with the service transport handle
svc_getreq()	Implements asynchronous event processing and returns control to the program after all sockets have been serviced
svc_register()	Registers procedures on the local Portmapper
svc_run()	Accepts RPC requests and calls the appropriate service using svc_getreq()
svc_sendreply()	Sends the results of an RPC to the caller
svc_socks[ ]	An array of socket descriptors being serviced
svc_unregister()	Removes all local mappings of *prognum versnum* to dispatch routines (*prognum*, *versnum,* *) and to port numbers
svcerr_auth()	Sends an error reply when the service dispatch routine cannot execute an RPC request because of authentication errors
svcerr_decode()	Sends an error reply when the service dispatch routine cannot decode its parameters
svcerr_noproc()	Sends an error reply when the service dispatch routine cannot call the procedure requested
svcerr_noprog()	Sends an error code when the requested program is not registered
svcerr_progvers()	Sends the low version number and high version number of RPC service when the version numbers of two RPC programs do not match
svcerr_systemerr()	Sends an error reply when the service dispatch routine detects a system error that has not been handled
svcerr_weakauth()	Sends an error reply when the service dispatch routine cannot run an RPC because of weak authentication parameters
svcraw_create()	Creates a local RPC service transport handle to simulate RPC programs within one host
svctcp_create()	Creates a TCP-based service transport
svcudp_create()	Creates a UDP-based service transport
xdr_accepted_reply()	Translates between an RPC reply message and its external representation
xdr_array()	Translates between an array and its external representation

xdr_authunix_parms()	Translates between UNIX-based authentication information and its external representation
xdr_bool()	Translates between a Boolean and its external representation
xdr_bytes()	Translates between byte strings and their external representations
xdr_callhdr()	Translates between an RPC message header and its external representation
xdr_callmsg()	Translates between RPC call messages (header and authentication, not argument data) and their external representations
xdr_destroy()	Destroys the XDR stream pointed to by the *xdrs* parameter
xdr_double()	Translates between C double-precision numbers and their external representations
xdr_enum()	Translates between C-enumerated groups and their external representations
xdr_float()	Translates between C floating-point numbers and their external representations
xdr_getpos()	Starts the get-position routine associated with the XDR stream, *xdrs*
xdr_inline()	Returns a pointer to a continuous piece of the XDR stream's buffer
xdr_int()	Translates between C integers and their external representations
xdr_long()	Translates between C long integers and their external representations
xdr_opaque()	Translates between fixed-size opaque data and its external representation
xdr_opaque_auth()	Translates between RPC message authentications and their external representations
xdr_pmap()	Translates an RPC procedure identification, such as is used in calls to Portmapper
xdr_pmaplist()	Translates a variable number of RPC procedure identifications, such as those Portmapper creates
xdr_reference()	Provides pointer chasing within structures
xdr_rejected_reply()	Translates between rejected RPC reply messages and their external representations
xdr_replymsg()	Translates between RPC reply messages and their external representations
xdr_setpos()	Starts the set-position routine associated with a XDR stream,

	xdrs
xdr_short()	Translates between C short integers and their external representations
xdr_string()	Translates between C strings and their external representations
xdr_u_int()	Translates between C unsigned integers and their external representations
xdr_u_long()	Translates between C unsigned long integers and their external representations
xdr_u_short()	Translates between C unsigned short integers and their external representations
xdr_union()	Translates between a discriminated C union and its external representation
xdr_vector()	Translates between a fixed-length array and its external representation
xdr_void()	Returns a value of 1
xdr_wrapstring()	Translates between strings and their external representations
xdrmem_create()	Initializes the XDR stream pointed to by *xdrs*
xdrrec_create()	Initializes the XDR stream pointed to by *xdrs*
xdrrec_endofrecord()	Marks the data in the output buffer as a completed record
xdrrec_eof()	Marks the end of the file, after using the rest of the current record in the XDR stream
xdrrec_skiprecord()	Discards the rest of the XDR stream's current record in the input buffer
xdrstdio_create()	Initializes the XDR stream pointed to by *xdrs*
xprt_register()	Registers service transport handles with the RPC service package; also modifies the global variable svc_socks[ ]
xprt_unregister()	Unregisters the RPC service transport handle

----

auth_destroy()

The auth_destroy() call destroys authentication information.

Syntax

```
#include <rpc\rpc.h>

void
auth_destroy(auth)
AUTH *auth;
```

Parameters

auth
Pointer to authentication information.

Description

The auth_destroy() call deletes the authentication information for *auth*. After you call this procedure, *auth* is undefined.

Related Calls

authnone_create()
authunix_create()
authunix_create_default()

----

authnone_create()

The authnone_create() call creates and returns a NULL RPC authentication handle.

Syntax

```
#include <rpc\rpc.h>

AUTH *
authnone_create()
```

Description

The authnone_create() call creates and returns an RPC authentication handle. The handle passes the NULL authentication on each call.

Related Calls

auth_destroy()
authunix_create()
authunix_create_default()

----

authunix_create()

The authunix_create() call creates and returns a UNIX-based authentication handle.

Syntax

```
#include <rpc\rpc.h>

AUTH *
authunix_create(host, uid, gid, len, aup_gids)
char *host;
int uid;
int gid;
int len;
int *aup_gids;
```

Parameters

host
Pointer to the symbolic name of the host where the desired server is located.

uid
User's user ID.

gid
User's group ID.

len
Length of the information pointed to by *aup_gids* .

aup_gids
Pointer to an array of groups to which the user belongs.

Description

The authunix_create() call creates and returns an authentication handle that contains UNIX-based authentication information.

Related Calls

auth_destroy()
authnone_create()
authunix_create_default()

----

authunix_create_default()

The authunix_create_default() call calls authunix_create() with default parameters.

Syntax

```
#include <rpc\rpc.h>

AUTH *
authunix_create_default()
```

Description

The authunix_create_default() call calls authunix_create() with default parameters.

Related Calls

auth_destroy()
authnone_create()
authunix_create()

----

callrpc()

The callrpc() call calls remote procedures.

Syntax

```
#include <rpc\rpc.h>
```

```
enum clnt_stat
callrpc(host, prognum, versnum, procnum, inproc, in, outproc, out)
char *host;
u_long prognum;
u_long versnum;
u_long procnum;
xdrproc_t inproc;
char *in;
xdrproc_t outproc;
char *out;
```

Parameters

host
Pointer to the symbolic name of the host where the desired server is located.

prognum
Program number of the remote procedure.

versnum
Version number of the remote procedure.

procnum
Procedure number of the remote procedure.

inproc
XDR procedure used to encode the arguments of the remote procedure.

in
Pointer to the arguments of the remote procedure.

outproc
XDR procedure used to decode the results of the remote procedure.

out
Pointer to the results of the remote procedure.

Description

The callrpc() call calls the remote procedure described by *prognum* , *versnum* , and *procnum* running on the *host* system. It encodes and decodes the parameters for transfer.

Notes:

1. You can use clnt_perrno() to translate the return code into messages.

2. callrpc() cannot call the procedure xdr_enum. See xdr_enum() for more information.

3. This procedure uses UDP as its transport layer. See clntudp_create() for more information.

Return Values

RPC_SUCCESS indicates success; otherwise, an error has occurred. The results of the remote procedure call return to *out* .

Examples

```
#define RMTPROGNUM (u_long)0x3fffffffL
#define RMTPROGVER (u_long)0x1
#define RMTPROCNUM (u_long)0x1

int inproc=100, outproc, rstat;
...
/* service request to host RPCSERVER_HOST */
if (rstat = callrpc("RPCSERVER_HOST", RMTPROGNUM, RMTPROGVER, RMTPROCNUM,
                    xdr_int, (char *)&inproc, xdr_int,
                    (char *)&outproc)!= 0)
    {
     clnt_perrno(rstat);
     exit(1);
    }
...
```

Related Calls

clnt_call()
clnt_perrno()
clntudp_create()

----

clnt_broadcast()

The clnt_broadcast() call broadcasts a remote program to all locally connected broadcast networks.

Syntax

```
#include <rpc\rpc.h>

enum clnt_stat
clnt_broadcast(prognum, versnum, procnum, inproc, in, outproc, out, eachresult)
u_long prognum;
u_long versnum;
u_long procnum;
xdrproc_t inproc;
caddr_t in;
xdrproc_t outproc;
caddr_t out;
resultproc_t eachresult;
```

Parameters

prognum
Program number of the remote procedure.

versnum
Version number of the remote procedure.

procnum
Procedure number of the remote procedure.

inproc
XDR procedure used to encode the arguments of the remote procedure.

in
Pointer to the arguments of the remote procedure.

outproc
XDR procedure used to decode the results of the remote procedure.

out
Pointer to the results of the remote procedure.

eachresult
Procedure called after each response.

Note: resultproc_t is a type definition:

```
   typedef bool_t (*resultproc_t) ();
```

Description

The clnt_broadcast() call broadcasts a remote program described by *prognum* , *versnum* , and *procnum* to all locally connected broadcast networks. Each time clnt_broadcast() receives a response, it calls eachresult().

The syntax and parameters of eachresult() are:

```
#include <netinet\in.h>
#include <rpc\rpctypes.h>

bool_t eachresult(out, addr)
char *out;
struct sockaddr_in *addr;
```

out

Has the same function as it does for clnt_broadcast(), except that the output of the remote procedure is decoded

addr

Pointer to the address of the machine that sent the results

Return Values

If eachresult() returns 0, clnt_broadcast() waits for more replies; otherwise, eachresult() returns the appropriate status.

Note: Broadcast sockets are limited in size to the maximum transfer unit of the data link.

Examples

```
enum clnt_stat cs;
u_long prognum, versnum;
...
cs = clnt_broadcast(prognum, versnum, NULLPROC, xdr_void,
                    (char *)NULL, xdr_void, (char *)NULL, eachresult);
if ((cs != RPC_SUCCESS) && (cs != RPC_TIMEDOUT))
  {
   fprintf( " broadcast failed: \n");
   exit(-1);
  }
...
bool_t
eachresult(out, addr)
void *out;                                  /* Nothing comes back */
struct sockaddr_in *addr;                      /* Reply from whom */
{
    register struct hostent *hp;
    ...
    hp = gethostbyaddr((char *) &addr->sin_addr, sizeof addr->sin_addr,
        AF_INET);
    printf("%s %s\n", inet_ntoa(addr->sin_addr), hp->h_name);
    ...
    return(FALSE);
}
```

Related Calls

callrpc()
clnt_call()

----

clnt_call()

The clnt_call() call calls the remote procedure (*procnum*) associated with the client handle (*clnt*).

Syntax

```
#include <rpc\rpc.h>

enum clnt_stat
clnt_call(clnt, procnum, inproc, in, outproc, out, tout)
CLIENT *clnt;
```

```
u_long procnum;
xdrproc_t inproc;
char *in;
xdrproc_t outproc;
char *out;
struct timeval tout;
```

Parameters

clnt
: Pointer to a client handle that was previously obtained using clntraw_create(), clnttcp_create(), or clntudp_create().

procnum
: Remote procedure number.

inproc
: XDR procedure used to encode *procnum*'s arguments.

in
: Pointer to the remote procedure's arguments.

outproc
: XDR procedure used to decode the remote procedure's results.

out
: Pointer to the remote procedure's results.

tout
: Time allowed for the server to respond, in units of 0.1 seconds.

Return Values

RPC_SUCCESS indicates success; otherwise, an error has occurred. The results of the remote procedure call are returned to *out*.

Examples

```
u_long procnum;
register CLIENT *clnt;
enum clnt_stat cs;
struct timeval  total_timeout;
int intsend, intrecv;

cs=clnt_call(clnt, procnum, xdr_int, &intsend,
   xdr_int, &intrecv, total_timeout);
if ( cs != RPC_SUCCESS)
    printf("*Error* clnt_call fail :\n");
```

Related Calls

- callrpc()
- clnt_perror()
- clntraw_create()
- clnttcp_create()
- clntudp_create()

----

clnt_destroy()

The clnt_destroy() call destroys a client's RPC handle.

Syntax

```
#include <rpc\rpc.h>

void
```

```
clnt_destroy(clnt)
CLIENT *clnt;
```

Parameters

clnt
Pointer to a client handle that was previously created using clntudp_create(), clnttcp_create(), or clntraw_create().

Description

The clnt_destroy() call deletes a client RPC transport handle. This procedure involves the deallocation of private data resources, including *clnt*. After you use this procedure, *clnt* is undefined. Open sockets associated with *clnt* must be closed.

Related Calls

clntraw_create()
clnttcp_create()
clntudp_create()

----

clnt_freeres()

The clnt_freeres() call deallocates resources assigned for decoding the results of an RPC.

Syntax

```
#include <rpc\rpc.h>

bool_t
clnt_freeres(clnt, outproc, out)
CLIENT *clnt;
xdrproc_t outproc;
char *out;
```

Parameters

clnt
Pointer to a client handle that was previously obtained using clntraw_create(), clnttcp_create(), or clntudp_create().

outproc
XDR procedure used to decode the remote procedure's results.

out
Pointer to the results of the remote procedure.

Return Values

The value 1 indicates success; the value 0 indicates an error.

Related Calls

clntraw_create()
clnttcp_create()
clntudp_create()

----

clnt_geterr()

The clnt_geterr() call copies the error structure from a client's handle to the local structure.

Syntax

```
#include <rpc\rpc.h>

void
clnt_geterr(clnt, errp)
CLIENT *clnt;
struct rpc_err *errp;
```

Parameters

clnt
: Pointer to a client handle that was previously obtained using clntraw_create(), clnttcp_create(), or clntudp_create().

errp
: Pointer to the address into which the error structure is copied.

Examples

```
u_long procnum;
register CLIENT *clnt;
enum clnt_stat cs;
struct timeval  total_timeout;
int intsend = 100, intrecv;
struct rpc_err error;
...
total_timeout.tv_sec = 20;
total_timeout.tv_usec = 0;
...
cs=clnt_call(clnt, procnum, xdr_int, &intsend,
   xdr_int, &intrecv, total_timeout);
if ( cs != RPC_SUCCESS)
     {
           clnt_geterr(clnt, &error);
           clnt_perror(clnt, "recv from server");
     }
...
```

Related Calls

clnt_call()
clnt_pcreateerror()
clnt_perrno()
clnt_perror()
clntraw_create()
clnttcp_create()
clntudp_create()

----

clnt_pcreateerror()

The clnt_pcreateerror() call indicates why a client handle cannot be created.

Syntax

```
#include <rpc\rpc.h>

void
clnt_pcreateerror(s)
char *s;
```

Parameters

s

Pointer to a string that is to be printed in front of the message. The string is followed by a colon.

Description

The clnt_pcreateerror() call writes a message to the standard error device, indicating why a client handle cannot be created. Use this procedure after the clntraw_create(), clnttcp_create(), or clntudp_create() call fails.

For an example of the clnt_pcreateerror() call, see clnttcp_create().

Related Calls

- clnt_geterr()
- clnt_perrno()
- clnt_perror()
- clntraw_create()
- clnttcp_create()
- clntudp_create()

----

clnt_perrno()

The clnt_perrno() call writes a message to the standard error device corresponding to the condition indicated by *stat*.

Syntax

```
#include <rpc\rpc.h>

void
clnt_perrno(stat)
enum clnt_stat stat;
```

Parameters

stat
The client status.

Description

The clnt_perrno() call writes a message to the standard error device corresponding to the condition indicated by *stat*. Use this procedure after callrpc() and clnt_broadcast() if there is an error.

Related Calls

- callrpc()
- clnt_geterr()
- clnt_pcreateerror()
- clnt_perror()

----

clnt_perror()

The clnt_perror() call writes an error message indicating why RPC failed.

Syntax

```
#include <rpc\rpc.h>

void
clnt_perror(clnt, s)
CLIENT *clnt;
```

```
char *s;
```

Parameters

clnt
: Pointer to a client handle that was previously obtained using clntudp_create(), clnttcp_create(), or clntraw_create().

s
: Pointer to a string that is to be printed in front of the message. The string is followed by a colon.

Description

The clnt_perror() call writes a message to the standard error device, indicating why an RPC failed. Use this procedure after clnt_call() if there is an error.

For an example of the clnt_perror() call, see clnt_geterr().

Related Calls

clnt_call()
clnt_geterr()
clnt_pcreateerror()
clnt_perrno()
clntraw_create()
clnttcp_create()
clntudp_create()

----

clntraw_create()

The clntraw_create() call creates a client transport handle to use in a single task.

Syntax

```
#include <rpc\rpc.h>

CLIENT *
clntraw_create(prognum, versnum)
u_long prognum;
u_long versnum;
```

Parameters

prognum
: Remote program number.

versnum
: Version number of the remote program.

Description

The clntraw_create() call creates a dummy client for the remote double (*prognum, versnum*). Because messages are passed using a buffer within the address space of the local process, the server should also use the same address space, which simulates RPC programs within one address space. See svcraw_create() for more information.

Return Values

NULL indicates failure.

Related Calls

clnt_call()
clnt_destroy()
clnt_pcreateerror()
clnttcp_create()

clntudp_create()
svcraw_create()

---

clnttcp_create()

The clnttcp_create() call creates an RPC client transport handle for the remote program using TCP transport.

Syntax

```
#include <rpc\rpc.h>

CLIENT *
clnttcp_create(addr, prognum, versnum, sockp, sendsz, recvsz)
struct sockaddr_in *addr;
u_long prognum;
u_long versnum;
int *sockp;
u_int sendsz;
u_int recvsz;
```

Parameters

addr
: Pointer to the internet address of the remote program. If *addr* points to a port number of 0, *addr* is set to the port on which the remote program is receiving.

prognum
: Remote program number.

versnum
: Version number of the remote program.

sockp
: Pointer to the socket. If *sockp* is RPC_ANYSOCK, then this routine opens a new socket and sets *sockp* .

sendsz
: Size of the send buffer. Specify 0 to have clnttcp_create() pick a suitable default size.

recvsz
: Size of the receive buffer. Specify 0 to have clnttcp_create() pick a suitable default size.

Description

The clnttcp_create() call creates an RPC client transport handle for the remote program specified by (*prognum, versnum*). The client uses TCP as the transport layer.

Return Values

NULL indicates failure.

Examples

```
#define RMTPROGNUM   (u_long)0x3fffffffL
#define RMTPROGVER   (u_long)0x1L

register CLIENT *clnt;
int sock = RPC_ANYSOCK; /* can be also valid socket descriptor */
struct hostent *hp;
struct sockaddr_in server_addr;

/* get the internet address of RPC server */
if ((hp = gethostbyname("RPCSERVER_HOST") == NULL)
  {
   fprintf(stderr,"Can't get address for %s\n",argv[2]);
   exit (-1);
  }
```

```
bcopy(hp->h_addr, (caddr_t)&server_addr.sin_addr.s_addr, hp->h_length);
server_addr.sin_family = AF_INET;
server_addr.sin_port = 0;



/* create TCP handle */
if ((clnt = clnttcp_create(&server_addr, RMTPROGNUM, RMTPROGVER,
                            &sock, 1024*10, 1024*10)) == NULL)
   {
    clnt_pcreateerror("clnttcp_create");
    exit(-1);
   }
```

Related Calls

- clnt_destroy()
- clnt_pcreateerror()
- clntraw_create()
- clntudp_create()

----

clntudp_create()

The clntudp_create() call creates an RPC client transport handle for the remote program using UDP transport.

Syntax

```
#include <rpc\rpc.h>
#include <netdb.h>

CLIENT *
clntudp_create(addr, prognum, versnum, wait, sockp)
struct sockaddr_in *addr;
u_long prognum;
u_long versnum;
struct timeval wait;
int *sockp;
```

Parameters

addr
: Pointer to the internet address of the remote program. If *addr* points to a port number of 0, *addr* is set to the port on which the remote program is receiving. The remote PORTMAP service is used for this.

prognum
: Remote program number.

versnum
: Version number of the remote program.

wait
: Interval at which UDP resends the call request, until either a response is received or the call times out. Set the time-out length using the clnt_call() procedure.

sockp
: Pointer to the socket. If *sockp* is RPC_ANYSOCK, this routine opens a new socket and sets *sockp* .

Description

The clntudp_create() call creates a client transport handle for the remote program (*prognum*) with version (*versnum*). UDP is used as the transport layer.

Note: Do not use this procedure with procedures that use large arguments or return large results. UDP RPC messages can contain only 2K bytes of encoded data.

Return Values

NULL indicates failure.

Examples

```
#define RMTPROGNUM   (u_long)0x3fffffffL
#define RMTPROGVER   (u_long)0x1L

register CLIENT *clnt;
int sock = RPC_ANYSOCK; /* can be also valid socket descriptor */
struct hostent *hp;
struct timeval pertry_timeout;
struct sockaddr_in server_addr;

/* get the internet address of RPC server */
if ((hp = gethostbyname("RPC_HOST") == NULL)
  {
   fprintf(stderr,"Can't get address for %s\n",argv[2]);
   exit (-1);
  }

pertry_timeout.tv_sec = 3;
pertry_timeout.tv_usec = 0;
bcopy(hp->h_addr, (caddr_t)&server_addr.sin_addr.s_addr, hp->h_length);
server_addr.sin_family = AF_INET;
server_addr.sin_port = 0;

/* create UDP handle */
if ((clnt = clntudp_create(&server_addr, RMTPROGNUM, RMTPROGVER,
                                pertry_timeout, &sock)) == NULL)
   {
    clnt_pcreateerror("clntudp_create");
    exit(-1);
   }
```

Related Calls

clnt_destroy()
clnt_pcreateerror()
clntraw_create()
clnttcp_create()

----

get_myaddress()

The get_myaddress() call returns the local host's internet address.

Syntax

```
#include <rpc\rpc.h>

void
get_myaddress(addr)
struct sockaddr_in *addr;
```

Parameters

addr
Pointer to the location where the local internet address is placed.

Description

The get_myaddress() call puts the local host's internet address into *addr*. The port number (*addr->sin_port*) is set to htons (PMAPPORT), which is 111.

----

getrpcbyname()

The getrpcbyname() call returns an RPC program entry specified by a name in the RPC file.

Syntax

```
#include <rpcnetdb.h>

struct rpcent *getrpcbyname(name)
char   *name;
```

Parameters

name
Pointer to the specified RPC program.

Description

The getrpcbyname() call sequentially searches from the beginning of the TCPIP\ETC\RPC file until it finds a matching RPC program name or encounters EOF.

Return Values

The getrpcbyname() call returns a pointer to an object with the rpcent structure for the RPC program specified on the call. The rpcent structure is defined in the <RPC\RPCNETDB.H> header file and contains the following elements:

Element	**Description**
r_name	The name of the server for this RPC program
r_aliases	A zero terminated list of alternate names for the RPC program
r_number	The RPC program number for this service

The return value points to static data that later calls overwrite. A pointer to an rpcent structure indicates success. A NULL pointer indicates an error or EOF.

Related Calls

getrpcbynumber()
getrpcent()

----

getrpcbynumber()

The getrpcbynumber() call returns an RPC program entry specified by a number in the RPC file.

Syntax

```
#include <rpcnetdb.h>

struct rpcent *getrpcbynumber(number)
u_long number;
```

Parameters

number
RPC program number.

Description

The getrpcbynumber() call sequentially searches from the beginning of the TCPIP\ETC\RPC file until it finds a matching RPC program number or encounters EOF.

Return Values

The getrpcbynumber() call returns a pointer to an object with the rpcent structure for the RPC program specified on the call. The rpcent structure is defined in <RPC\RPCNETDB.H> header file and contains the following elements:

Element	**Description**
r_name	The name of the server for this RPC program
r_aliases	A zero terminated list of alternate names for the RPC program
r_number	The RPC program number for this service

The return value points to static data that later calls overwrite. A pointer to an rpcent structure indicates success. A NULL pointer indicates an error or EOF.

Related Calls

getrpcbyname()
getrpcent()

----

getrpcent()

The getrpcent() call returns the next entry in the TCPIP\ETC\RPC file.

Syntax

```
#include <rpcnetdb.h>

struct rpcent *getrpcent()
```

Return Values

The getrpcent() call returns a pointer to the next entry in the TCPIP\ETC\RPC file. The rpcent structure is defined in the <RPC\RPCNETDB.H> header file and contains the following elements:

Element	**Description**
r_name	The name of the server for this RPC program
r_aliases	A zero terminated list of alternate names for the RPC program
r_number	The RPC program number for this service

The return value points to static data that later calls overwrite. A pointer to an rpcent structure indicates success. A NULL pointer indicates an error or EOF.

Related Calls

getrpcbyname()
getrpcbynumber()

----

pmap_getmaps()

The pmap_getmaps() call returns a list of current program-to-port mappings on a specified remote host's Portmapper.

Syntax

```
#include <rpc\rpc.h>

struct pmaplist *
pmap_getmaps(addr)
struct sockaddr_in *addr;
```

Parameters

addr
Pointer to the internet address of the remote host.

Description

The pmap_getmaps() call returns a list of current program-to-port mappings on the remote host's Portmapper specified by *addr*.

Examples

```
struct hostent *hp;
struct sockaddr_in pmapper_addr;
struct pmaplist *my_pmaplist = NULL;

if ((hp = gethostbyname("PMAP_HOST") == NULL)
  {
   fprintf(stderr,"Can't get address for %s\n","PMAP_HOST");
   exit (-1);
  }

bcopy(hp->h_addr, (caddr_t)&pmapper_addr.sin_addr.s_addr, hp->h_length);
pmapper_addr.sin_family = AF_INET;
pmapper_addr.sin_port = 0;

/*
 *  get the list of program, version, protocol and port number
 *  from remote portmapper
 *
 *     struct pmap {
 *              long unsigned pm_prog;
 *              long unsigned pm_vers;
 *              long unsigned pm_prot;
 *              long unsigned pm_port;
 *             };



 *     struct pmaplist {
 *             struct pmap     pml_map;
 *             struct pmaplist *pml_next;
 *            };
 */
 my_pmaplist = pmap_getmaps(&pmapper_addr);
 ...
```

Related Calls

pmap_getport()
pmap_rmtcall()
pmap_set()
pmap_unset()

----

pmap_getport()

The pmap_getport() call returns the port number associated with the remote program (*prognum*), the version (*versnum*), and the transport protocol (*protocol*).

Syntax

```
#include <rpc\rpc.h>

u_short
pmap_getport(addr, prognum, versnum, protocol)
struct sockaddr_in *addr;
u_long prognum;
u_long versnum;
u_long protocol;
```

Parameters

addr
Pointer to the internet address of the remote host.

prognum
Program number to be mapped.

versnum
Version number of the program to be mapped.

protocol
Transport protocol used by the program.

Return Values

The value 0 indicates that the mapping does not exist or that the remote PORTMAP could not be contacted. If Portmapper cannot be contacted, *rpc_createerr* contains the RPC status.

Related Calls

pmap_getmaps()
pmap_rmtcall()
pmap_set()
pmap_unset()

----

pmap_rmtcall()

The pmap_rmtcall() call instructs Portmapper to make an RPC call to a procedure on a host on your behalf. Use this procedure only for ping-type functions.

Syntax

```
#include <rpc\rpc.h>
#include <netdb.h>

enum clnt_stat
pmap_rmtcall(addr, prognum, versnum, procnum , inproc, in,
             outproc, out, tout, portp)
struct sockaddr_in *addr;
u_long prognum;
u_long versnum;
u_long procnum;
xdrproc_t inproc;
char *in;
xdrproc_t outproc;
char *out;
struct timeval tout;
u_long *portp;
```

Parameters

addr

Pointer to the internet address of the foreign host.

prognum
Remote program number.

versnum
Version number of the remote program.

procnum
Procedure to be called.

inproc
XDR procedure that encodes the arguments of the remote procedure.

in
Pointer to the arguments of the remote procedure.

outproc
XDR procedure that decodes the results of the remote procedure.

out
Pointer to the results of the remote procedure.

tout
Time-out period for the remote request.

portp
Port number of the triple (*prognum* , *versnum* , *procnum*), if the call from the remote PORTMAP service succeeds.

Return Values

RPC_SUCCESS indicates success; otherwise, an error has occurred. The results of the remote procedure call return to *out* .

Examples

```
int inproc, outproc,rc;
u_long portp;
struct timeval total_timeout;
struct sockaddr_in *addr;
...
get_myaddress(addr);
...
total_timeout.tv_sec = 20;
total_timeout.tv_usec = 0;

rc = pmap_rmtcall(addr,RMTPROGNUM,RMTPROGVER,RMTPROCNUM,xdr_int,
&inproc,xdr_int,&outproc,total_timeout,&portp);
if (rc != 0)
 {
  fprintf(stderr,"error: pmap_rmtcall() failed: %d \n",rc);
  clnt_perrno(rc);
  exit(1);
 }
```

Related Calls

pmap_getmaps()
pmap_getport()
pmap_set()
pmap_unset()

----

pmap_set()

The pmap_set() call sets the mapping of the program (specified by *prognum* , *versnum* , and *protocol*) to *port* on the local machine's Portmapper. This procedure is automatically called by the svc_register() procedure.

Syntax

```
#include <rpc\rpc.h>

bool_t
pmap_set(prognum, versnum, protocol, port)
u_long prognum;
u_long versnum;
u_long protocol;
u_short port;
```

Parameters

prognum
Local program number.

versnum
Version number of the local program.

protocol
Transport protocol used by the local program.

port
Port to which the local program is mapped.

Return Values

The value 1 indicates success; the value 0 indicates an error.

Related Calls

pmap_getmaps()
pmap_getport()
pmap_rmtcall()
pmap_unset()

----

pmap_unset()

The pmap_unset() call removes the mappings associated with *prognum* and *versnum* on the local machine's Portmapper. All ports for each transport protocol currently mapping the *prognum* and *versnum* are removed from the PORTMAP service.

Syntax

```
#include <rpc\rpc.h>

bool_t
pmap_unset(prognum, versnum)
u_long prognum;
u_long versnum;
```

Parameters

prognum
Local program number.

versnum
Version number of the local program.

Return Values

The value 1 indicates success; the value 0 indicates an error.

Examples

```
#define RMTPROGNUM   (u_long)0x3fffffffL
#define RMTPROGVER   (u_long)0x1L
...
/* remove the mapping of remote program */
/* and its port from local Portmapper   */
pmap_unset(RMTPROGNUM, RMTPROGVER);
...
```

Related Calls

pmap_getmaps()
pmap_getport()
pmap_rmtcall()
pmap_set()

----

registerrpc()

The registerrpc() call registers a procedure with the local Portmapper and creates a control structure to remember the server procedure and its XDR routine. The svc_run() call uses the control structure. Procedures registered using registerrpc() are accessed using the UDP transport layer.

Syntax

```
#include <rpc\rpc.h>

int
registerrpc(prognum, versnum, procnum, procname, inproc, outproc)
u_long prognum;
u_long versnum;
u_long procnum;
char *(*procname) ();
xdrproc_t inproc;
xdrproc_t outproc;
```

Parameters

prognum
Program number to register.

versnum
Version number to register.

procnum
Procedure number to register.

procname
Procedure that is called when the registered program is requested. *procname* must accept a pointer to its arguments and return a static pointer to its results.

inproc
XDR procedure that decodes the arguments.

outproc
XDR procedure that encodes the results.

Note: You cannot use xdr_enum() as an argument to registerrpc(). See xdr_enum() for more information.

Return Values

The value 0 indicates success; the value -1 indicates an error.

Examples

```
#define RMTPROGNUM (u_long)0x3fffffffL
#define RMTPROGVER (u_long)0x1
#define RMTPROCNUM (u_long)0x1

main()
 {
  int *rmtprog();

  /* register remote program with Portmapper */
  registerrpc(RMTPROGNUM, RMTPROGVER, RMTPROCNUM, rmtprog,
  xdr_int, xdr_int);

  /* infinite loop, waits for RPC request from client */
  svc_run();
  printf("Error: svc_run should never reach this point \n");
  exit(1);
 }

int *
rmtprog(inproc)          /* remote program */
int *inproc;

{
 int *outproc;
 ...
 /* Process request */
 ...
 return (outproc);
}
```

Related Calls

svc_register()
svc_run()

----

rpc_createerr

The rpc_createerr global variable is set when any RPC client creation routine fails. Use clnt_pcreateerror() to print the message.

Syntax

```
#include <rpc\rpc.h>

struct  rpc_createerr rpc_createerr;
```

----

svc_destroy()

The svc_destroy() call deletes the RPC service transport handle *xprt*, which becomes undefined after this routine is called.

Syntax

```
#include <rpc\rpc.h>

void
svc_destroy(xprt)
SVCXPRT *xprt;
```

Parameter

xprt
    Pointer to the service transport handle.

Related Calls

svcraw_create()
svctcp_create()
svcudp_create()

----

svc_freeargs()

The svc_freeargs() call frees storage allocated to decode the arguments received by svc_getargs().

Syntax

```
#include <rpc\rpc.h>

bool_t
svc_freeargs(xprt, inproc, in)
SVCXPRT *xprt;
xdrproc_t inproc;
char *in;
```

Parameters

xprt
    Pointer to the service transport handle.

inproc
    XDR routine that decodes the arguments.

in
    Pointer to the input arguments.

Return Values

The value 1 indicates success; the value 0 indicates an error.

Related Calls

svc_getargs()

----

svc_getargs()

The svc_getargs() call uses the XDR routine *inproc* to decode the arguments of an RPC request associated with the RPC service transport handle *xprt*. The results are placed at address *in*.

Syntax

```
#include <rpc\rpc.h>

bool_t
svc_getargs(xprt, inproc, in)
SVCXPRT *xprt;
xdrproc_t inproc;
char *in;
```

Parameters

xprt
Pointer to the service transport handle.

inproc
XDR routine that decodes the arguments.

in
Pointer to the decoded arguments.

Return Values

The value 1 indicates success; the value 0 indicates an error.

Examples

```
#define RMTPROGNUM   (u_long)0x3fffffffL
#define RMTPROGVER   (u_long)0x1L

...

SVCXPRT *transp;

transp = svcudp_create(RPC_ANYSOCK);
if (transp == NULL)
   {
    fprintf(stderr, "can't create an RPC server transport\n");
    exit(-1);
   }
pmap_unset(RMTPROGNUM, RMTPROGVER);
if (!svc_register(transp, RMTPROGNUM, RMTPROGVER, rmtprog, IPPROTO_UDP))
   {
    fprintf(stderr, "can't register rmtprog() service\n");
    exit(-1);
   }
printf("rmtprog() service registered.\n");

svc_run();
printf("Error:svc_run should never reach this point \n");
exit(1);
...



rmtprog(rqstp, transp)
struct svc_req *rqstp;
SVCXPRT *transp;
{
    int intrecv;

    switch((int)rqstp->rq_proc)
       {
           case PROCNUM1:
                    svc_getargs(transp, xdr_int, &intrecv);
                      ...
                    return;
           case PROCNUM2:
             ...
       }
...
}
```

Related Calls

svc_freeargs()

----

svc_getcaller()

The svc_getcaller() call gets the network address of the client associated with the service transport handle.

Syntax

```
#include <rpc\rpc.h>

struct sockaddr_in *
svc_getcaller(xprt)
SVCXPRT *xprt;
```

Parameters

xprt
: Pointer to the service transport handle.

----

svc_getreq()

The svc_getreq() call implements asynchronous event processing and returns control to the program after all sockets have been serviced.

Syntax

```
#include <rpc\rpc.h>

void
svc_getreq(socks, noavail)
int socks[];
int noavail;
```

Parameters

socks
: Array of socket descriptors.

noavail
: Integer specifying the number of socket descriptors in the array.

Description

Use the svc_getreq() call rather than svc_run() to do asynchronous event processing. The routine returns control to the program when all sockets in the *socks* array have been serviced.

Related Calls

svc_run()
svc_socks[ ]

----

svc_register()

The svc_register() call registers procedures on the local Portmapper.

Syntax

```
#include <rpc\rpc.h>
#include <netdb.h>
```

```
bool_t
svc_register(xprt, prognum, versnum, dispatch, protocol)
SVCXPRT *xprt;
u_long prognum;
u_long versnum;
void (*dispatch) ();
int protocol;
```

Parameters

xprt
Pointer to the service transport handle.

prognum
Program number to be registered.

versnum
Version number of the program to be registered.

dispatch
Dispatch routine associated with *prognum* and *versnum* . The structure of the dispatch routine is as follows:

```
dispatch(request, xprt)
struct svc_req *request;
SVCXPRT *xprt;
```

protocol
Protocol used. The value is generally one of the following:

- 0 (zero)
- IPPROTO_UDP
- IPPROTO_TCP

When you use a value of 0, the service is not registered with Portmapper.

Description

The svc_register() call associates the specified program with the service dispatch routine *dispatch* .

Note: When you use a toy RPC service transport created with svcraw_create(), make a call to xprt_register() immediately after a call to svc_register().

Return Values

The value 1 indicates success; the value 0 indicates an error.

Examples

```
#define RMTPROGNUM   (u_long)0x3fffffffL
#define RMTPROGVER   (u_long)0x1L

SVCXPRT *transp;

/* register the remote program with local Portmapper */
if (!svc_register(transp, RMTPROGNUM, RMTPROGVER, rmtprog, IPPROTO_UDP))
       {
        fprintf(stderr, "can't register rmtprog() service\n");
        exit(-1);
       }



/* code for remote program; rmtprog  */
rmtprog(rqstp, transp)
struct svc_req *rqstp;
SVCXPRT *transp;
  {
     ...
  }
```

Related Calls

registerrpc()
svc_unregister()
xprt_register()

----

svc_run()

The svc_run() call accepts RPC requests and calls the appropriate service using svc_getreq(). The svc_run() call does not return control to the caller.

Syntax

```
#include <rpc\rpc.h>

void
svc_run()
```

Examples

```
#define RMTPROGNUM   (u_long)0x3fffffffL
#define RMTPROGVER   (u_long)0x1L

...

SVCXPRT *transp;

transp = svcudp_create(RPC_ANYSOCK);
if (transp == NULL)
   {
    fprintf(stderr, "can't create an RPC server transport\n");
    exit(-1);
   }
pmap_unset(RMTPROGNUM, RMTPROGVER);
if (!svc_register(transp, RMTPROGNUM, RMTPROGVER, rmtprog, IPPROTO_UDP))
   {
    fprintf(stderr, "can't register rmtprog() service\n");
    exit(-1);
   }
printf("rmtprog() service registered.\n");

svc_run();

printf("Error:svc_run should never reach this point \n");
exit(1);
...



rmtprog(rqstp, transp)
struct svc_req *rqstp;
SVCXPRT *transp;
 {
 ...
 }
```

Related Calls

registerrpc()
svc_getreq()

----

svc_sendreply()

The svc_sendreply() call sends the results of an RPC to the caller.

Syntax

```
#include <rpc\rpc.h>

bool_t
svc_sendreply(xprt, outproc, out)
SVCXPRT *xprt;
xdrproc_t outproc;
char *out;
```

Parameters

xprt
Pointer to the caller's transport handle.

outproc
XDR procedure that encodes the results.

out
Pointer to the results.

Description

The service dispatch routine calls the svc_sendreply() call to send the results of the call to the caller.

Return Values

The value 1 indicates success; the value 0 indicates an error.

Examples

```
#define RMTPROGNUM   (u_long)0x3fffffffL
#define RMTPROGVER   (u_long)0x1L

...

SVCXPRT *transp;

transp = svcudp_create(RPC_ANYSOCK);
if (transp == NULL)
   {
    fprintf(stderr, "can't create an RPC server transport\n");
    exit(-1);
   }
pmap_unset(RMTPROGNUM, RMTPROGVER);
if (!svc_register(transp, RMTPROGNUM, RMTPROGVER, rmtprog, IPPROTO_UDP))
   {
    fprintf(stderr, "can't register rmtprog() service\n");
    exit(-1);
   }
printf("rmtprog() service registered.\n");

svc_run();

printf("Error:svc_run should never reach this point \n");
exit(1);
...




rmtprog(rqstp, transp)
struct svc_req *rqstp;
SVCXPRT *transp;
{

    int intrecv;
    int replysend;
    switch((int)rqstp->rq_proc)
     {
        case PROCNUM0:
           svc_getargs(transp, xdr_int, &intrecv);
                      ...
           /* process intrecv parameter */
           replysend = ( intrecv * 1000) + 100;
           /*  send reply to client */
           if (!svc_sendreply(transp, xdr_int, &replysend))
              {
               fprintf(stderr,"can't reply to RPC call\n");
               exit(-1);
              }
           return;
        case PROCNUM1:
           ...
     }
...
}
```

----

svc_socks[ ]

The array svc_socks[ ] is an array of socket descriptors being serviced. The integer *noregistered* specifies the number of socket descriptors in svc_socks[ ].

Syntax

```
#include <rpc\rpc.h>

int svc_socks[ ];
```

```
#include <rpc\rpc.h>

int noregistered;
```

Related Calls

svc_getreq()

----

svc_unregister()

The svc_unregister() call removes all local mappings of *prognum versnum* to dispatch routines (*prognum* , *versnum,* *) and to port numbers.

Syntax

```
#include <rpc\rpc.h>

void
svc_unregister(prognum, versnum)
u_long prognum;
u_long versnum;
```

Parameters

prognum

Program number of the removed program.

versnum
Version number of the removed program.

Examples

```
#define RMTPROGNUM   (u_long)0x3fffffffL
#define RMTPROGVER   (u_long)0x1L
...
  /* unregister remote program from local Portmapper */
  svc_unregister(RMTPROGNUM, RMTPROGVER);
...
```

Related Calls

svc_register()

----

svcerr_auth()

The svcerr_auth() call sends an error reply when the service dispatch routine cannot execute an RPC request because of authentication errors.

Syntax

```
#include <rpc\rpc.h>

void
svcerr_auth(xprt, why)
SVCXPRT *xprt;
enum auth_stat why;
```

Parameters

xprt
Pointer to the service transport handle.

why
Reason why the call is refused.

Description

A service dispatch routine that refuses to run an RPC request because of authentication errors calls svcerr_auth().

Related Calls

svcerr_decode()
svcerr_noproc()
svcerr_noprog()
svcerr_progvers()
svcerr_systemerr()
svcerr_weakauth()

----

svcerr_decode()

The svcerr_decode() call sends an error reply when the service dispatch routine cannot decode its parameters.

Syntax

```
#include <rpc\rpc.h>

void
svcerr_decode(xprt)
SVCXPRT *xprt;
```

Parameters

xprt
Pointer to the service transport handle.

Description

A service dispatch routine that cannot decode its parameters calls svcerr_decode().

Related Calls

svcerr_auth()
svcerr_noproc()
svcerr_noprog()
svcerr_progvers()
svcerr_systemerr()
svcerr_weakauth()

----

svcerr_noproc()

The svcerr_noproc() call sends an error reply when the service dispatch routine cannot call the procedure requested.

Syntax

```
#include <rpc\rpc.h>

void
svcerr_noproc(xprt)
SVCXPRT *xprt;
```

Parameters

xprt
Pointer to the service transport handle.

Description

A service dispatch routine that does not implement the requested procedure calls the svcerr_noproc() call.

Related Calls

svcerr_auth()
svcerr_decode()
svcerr_noprog()
svcerr_progvers()
svcerr_systemerr()
svcerr_weakauth()

----

svcerr_noprog()

The svcerr_noprog() call sends an error code when the requested program is not registered.

Syntax

```
#include <rpc\rpc.h>

void
svcerr_noprog(xprt)
SVCXPRT *xprt;
```

Parameters

xprt
Pointer to the service transport handle.

Description

Use the svcerr_noprog() call when the desired program is not registered.

Related Calls

svcerr_auth()
svcerr_decode()
svcerr_noproc()
svcerr_progvers()
svcerr_systemerr()
svcerr_weakauth()

----

svcerr_progvers()

The svcerr_progvers() call sends the low version number and high version number of RPC service when the version numbers of two RPC programs do not match.

Syntax

```
#include <rpc\rpc.h>

void
svcerr_progvers(xprt, low_vers, high_vers)
SVCXPRT *xprt;
u_long low_vers;
u_long high_vers;
```

Parameters

xprt
Pointer to the service transport handle.

low_vers
Low version number.

high_vers
High version number.

Description

A service dispatch routine calls the svcerr_progvers() call when the version numbers of two RPC programs do not match. The call sends the supported low version and high version of RPC service.

Related Calls

svcerr_decode()
svcerr_noproc()
svcerr_noprog()

svcerr_systemerr()
svcerr_weakauth()

---

svcerr_systemerr()

The svcerr_systemerr() call sends an error reply when the service dispatch routine detects a system error that has not been handled.

Syntax

```
#include <rpc\rpc.h>

void
svcerr_systemerr(xprt)
SVCXPRT *xprt;
```

Parameters

xprt
Pointer to the service transport handle.

Description

A service dispatch routine calls the svcerr_systemerr() call when it detects a system error that is not handled by the protocol.

Related Calls

svcerr_auth()
svcerr_decode()
svcerr_noproc()
svcerr_noprog()
svcerr_progvers()
svcerr_weakauth()

---

svcerr_weakauth()

The svcerr_weakauth() call sends an error reply when the service dispatch routine cannot run an RPC because of weak authentication parameters.

Syntax

```
#include <rpc\rpc.h>

void
svcerr_progvers(xprt)
SVCXPRT *xprt;
```

Parameters

xprt
Pointer to the service transport handle.

Description

A service dispatch routine calls the svcerr_weakauth() call when it cannot run an RPC because of correct but weak authentication parameters

Note: This is the equivalent of `svcerr_auth(xprt, AUTH_TOOWEAK).`

Related Calls

svcerr_auth()
svcerr_decode()
svcerr_noproc()
svcerr_noprog()
svcerr_progvers()
svcerr_systemerr()

----

svcraw_create()

The svcraw_create() call creates a local RPC service transport handle to simulate RPC programs within one host.

Syntax

```
#include <rpc\rpc.h>

SVCXPRT *
svcraw_create()
```

Description

The svcraw_create() call creates a local RPC service transport used for timings, to which it returns a pointer. Because messages are passed using a buffer within the address space of the local process, the client process must also use the same address space. This allows the simulation of RPC programs within one host. See clntraw_create() for more information.

Return Values

NULL indicates failure.

Related Calls

clntraw_create()
svc_destroy()
svctcp_create()
svcudp_create()

----

svctcp_create()

The svctcp_create() call creates a TCP-based service transport.

Syntax

```
#include <rpc\rpc.h>

SVCXPRT *
svctcp_create(sock, send_buf_size, recv_buf_size)
int sock;
u_int send_buf_size;
u_int recv_buf_size;
```

Parameters

sock
: Socket descriptor. If *sock* is RPC_ANYSOCK, a new socket is created. If the socket is not bound to a local TCP port, it is bound to an arbitrary port.

send_buf_size
Size of the send buffer. Specify 0 if you want the call to pick a suitable default value.

recv_buf_size
Size of the receive buffer. Specify 0 if you want the call to pick a suitable default value.

Description

The svctcp_create() call creates a TCP-based service transport to which it returns a pointer. *xprt*->xp_sock contains the transport's socket descriptor; *xprt*->xp_port contains the transport's port number.

Return Values

NULL indicates failure.

Examples

```
...
SVCXPRT *transp;

transp = svctcp_create(RPC_ANYSOCK, 1024*10, 1024*10);
...
```

Related Calls

svc_destroy()
svcraw_create()
svcudp_create()

----

svcudp_create()

The svcudp_create() call creates a UDP-based service transport.

Syntax

```
#include <rpc\rpc.h>

SVCXPRT *
svcudp_create(sockp)
int sockp;
```

Parameters

sockp
The socket number associated with the service transport handle. If *sockp* is RPC_ANYSOCK, a new socket is created. If the socket is not bound to a local port, it is bound to an arbitrary port.

Description

The svcudp_create() call creates a UDP-based service transport to which it returns a pointer. *xprt*->xp_sock contains the transport's socket descriptor. *xprt*->xp_port contains the transport's port number.

Return Values

NULL indicates failure.

Examples

```
...
SVCXPRT *transp;

transp = svcudp_create(RPC_ANYSOCK);
...
```

Related Calls

- svc_destroy()
- svcraw_create()
- svctcp_create()

----

xdr_accepted_reply()

The xdr_accepted_reply() call translates between an RPC reply message and its external representation.

Syntax

```
#include <rpc\rpc.h>

bool_t
xdr_accepted_reply(xdrs, ar)
XDR *xdrs;
struct accepted_reply *ar;
```

Parameters

xdrs
: Pointer to an XDR stream.

ar
: Pointer to the reply to be represented.

Return Values

The value 1 indicates success; the value 0 indicates an error.

----

xdr_array()

The xdr_array() call translates between an array and its external representation.

Syntax

```
#include <rpc\rpc.h>

bool_t
xdr_array(xdrs, arrp, sizep, maxsize, elsize, elproc)
XDR *xdrs;
char **arrp;
u_int *sizep;
u_int maxsize;
u_int elsize;
xdrproc_t elproc;
```

Parameters

xdrs
: Pointer to an XDR stream.

arrp
: Address of the pointer to the array.

sizep
Pointer to the element count of the array.

maxsize
Maximum number of elements accepted.

elsize
Size of each of the array's elements, found using sizeof().

elproc
XDR routine that translates an individual array element.

Return Values

The value 1 indicates success; the value 0 indicates an error.

Examples

```
struct myarray
   {
    int  *arrdata;
    u_int   arrlength;
   };

void
xdr_myarray(xdrsp,arrp)
XDR  *xdrsp;
struct myarray *arrp;
{
  xdr_array(xdrsp,(caddr_t *)&arrp->arrdata,&arrp->arrlength,
                                    MAXLEN,sizeof(int),xdr_int);
}

...
static int arrc_in[10],arrc_out[10];
...
u_long procnum;
register CLIENT *clnt;
enum clnt_stat cs;
struct timeval  total_timeout;
...
total_timeout.tv_sec = 20;
total_timeout.tv_usec = 0;
...
myarrc_in.arrdata =  & arrc_in&lbrk.0&rbrk.;
myarrc_in.arrlength = ( sizeof(arrc_in) / sizeof (int) );
myarrc_out.arrdata = & arrc_out&lbrk.0&rbrk.;
myarrc_out.arrlength = ( sizeof(arrc_out) / sizeof (int) );

cs=clnt_call(clnt, procnum, xdr_myarray, (char *) &myarrc_in, xdr_myarray,
                                 (char *)&myarrc_out, total_timeout);
if ( cs != RPC_SUCCESS)
        printf("*Error* clnt_call fail :\n");
...
```

----

xdr_authunix_parms()

The xdr_authunix_parms() call translates between UNIX-based authentication information and its external representation.

Syntax

```
#include <rpc\rpc.h>

bool_t
xdr_authunix_parms(xdrs, aupp)
XDR *xdrs;
struct authunix_parms *aupp;
```

Parameters

xdrs
Pointer to an XDR stream.

aupp
Pointer to the authentication information.

Return Values

The value 1 indicates success; the value 0 indicates an error.

----

xdr_bool()

The xdr_bool() call translates between a Boolean and its external representation.

Syntax

```
#include <rpc\rpc.h>

bool_t
xdr_bool(xdrs, bp)
XDR *xdrs;
bool_t *bp;
```

Parameters

xdrs
Pointer to an XDR stream.

bp
Pointer to the Boolean.

Return Values

The value 1 indicates success; the value 0 indicates an error.

----

xdr_bytes()

The xdr_bytes() call translates between byte strings and their external representations.

Syntax

```
#include <rpc\rpc.h>

bool_t
xdr_bytes(xdrs, sp, sizep, maxsize)
XDR *xdrs;
char **sp;
u_int *sizep;
u_int maxsize;
```

Parameters

xdrs
Pointer to an XDR stream.

sp
Pointer to a pointer to the byte string.

sizep
Pointer to the byte string size.

maxsize
Maximum size of the byte string.

Return Values

The value 1 indicates success; the value 0 indicates an error.

Examples

```
struct mybytes
   {
    char    *bytdata;
    u_int   bytlength;
   };

void
xdr_mybytes(xdrsp,arrp)
XDR  *xdrsp;
struct mybytes *arrp;
{
   xdr_bytes(xdrsp,(caddr_t *)&arrp->bytdata,&arrp->bytlength,MAXLEN);
}

...
char *bytc_in ,*bytc_out;
...
u_long procnum;
register CLIENT *clnt;
enum clnt_stat cs;
struct timeval  total_timeout;
...
total_timeout.tv_sec = 20;
total_timeout.tv_usec = 0;
...

mybytc_in.bytdata =  bytc_in;
mybytc_in.bytlength = strlen(bytc_in)+1;
cs=clnt_call(clnt, procnum, xdr_mybytes, (caddr_t *) &mybytc_in,
            xdr_mybytes, (caddr_t *)&mybytc_out, total_timeout);
if ( cs != RPC_SUCCESS)
   printf("*Error* clnt_call fail :\n");
```

----

xdr_callhdr()

The xdr_callhdr() call translates between an RPC message header and its external representation.

Syntax

```
#include <rpc\rpc.h>

void
xdr_callhdr(xdrs, chdr)
XDR *xdrs;
struct rpc_msg *chdr;
```

Parameters

xdrs
Pointer to the XDR stream.

chdr
    Pointer to the call header.

----

xdr_callmsg()

The xdr_callmsg() call translates between RPC call messages (header and authentication, not argument data) and their external representations.

Syntax

```
#include <rpc\rpc.h>

void
xdr_callmsg(xdrs, cmsg)
XDR *xdrs;
struct rpc_msg *cmsg;
```

Parameters

xdrs
    Pointer to the XDR stream.

cmsg
    Pointer to the call message.

----

xdr_destroy()

The xdr_destroy() call destroys the XDR stream pointed to by the *xdrs* parameter.

Syntax

```
#include <rpc\rpc.h>

void xdr_destroy(xdrs)
XDR *xdrs;
```

Parameters

xdrs
    Pointer to the XDR stream.

Description

The xdr_destroy() call invokes the destroy routine associated with the eXternal Data Representation stream pointed to by the *xdrs* parameter, and frees the private data structures allocated to the stream.

----

xdr_double()

The xdr_double() call translates between C double-precision numbers and their external representations.

Syntax

```
#include <rpc\rpc.h>

bool_t
xdr_double(xdrs, dp)
XDR *xdrs;
double *dp;
```

Parameters

xdrs
Pointer to the XDR stream.

dp
Pointer to a double-precision number.

Return Values

The value 1 indicates success; the value 0 indicates an error.

----

xdr_enum()

The xdr_enum() call translates between C-enumerated groups and their external representations.

Syntax

```
#include <rpc\rpc.h>

bool_t
xdr_enum(xdrs, ep)
XDR *xdrs;
enum_t *ep;
```

Parameters

xdrs
Pointer to the XDR stream.

ep
Pointer to the enumerated number.

Description

The xdr_enum() call translates between C-enumerated groups and their external representations. When you call the procedures callrpc() and registerrpc(), create a stub procedure for both the server and the client before the procedure of the application program using xdr_enum(). Verify that this procedure looks like the following:

```
#include <rpc\rpc.h>

void
static xdr_enum_t(xdrs, ep)
XDR *xdrs;
enum_t *ep;
{
       xdr_enum(xdrs, ep)
}
```

The xdr_enum_t procedure is used as the *inproc* and *outproc* in both the client and server RPCs.

For example, an RPC client would contain the following lines:

```
                         ...

error = callrpc(argv[1],ENUMRCVPROG,VERSION,ENUMRCVPROC,
                     xdr_enum_t,&innumber,xdr_enum_t,&outnumber);
```

```
                              ...
```

An RPC server would contain the following line:

```
registerrpc(ENUMRCVPROG,VERSION,ENUMRCVPROC,xdr_enum_t,
                    xdr_enum_t);

                              ...
```

Return Values

The value 1 indicates success; the value 0 indicates an error.

----

xdr_float()

The xdr_float() call translates between C floating-point numbers and their external representations.

Syntax

```
#include <rpc\rpc.h>

bool_t
xdr_float(xdrs, fp)
XDR *xdrs;
float *fp;
```

Parameters

xdrs
  Pointer to the XDR stream.

fp
  Pointer to the floating-point number.

Return Values

The value 1 indicates success; the value 0 indicates an error.

----

xdr_getpos()

The xdr_getpos() call starts the get-position routine associated with the XDR stream, *xdrs* .

Syntax

```
#include <rpc\rpc.h>

u_int
xdr_getpos(xdrs)
XDR *xdrs;
```

Parameters

xdrs
  Pointer to the XDR stream.

Return Values

The xdr_getpos() call returns an unsigned integer, which indicates the position of the XDR byte stream.

Related Calls

xdr_setpos()

----

xdr_inline()

The xdr_inline() call returns a pointer to a continuous piece of the XDR stream's buffer.

Syntax

```
#include <rpc\rpc.h>

long *
xdr_inline(xdrs, len)
XDR *xdrs;
int len;
```

Parameters

xdrs
Pointer to the XDR stream.

len
Length in bytes of the desired buffer.

Description

The xdr_inline() call returns a pointer to a continuous piece of the XDR stream's buffer. The value is `long *` rather than `char *`, because the external data representation of any object is always an integer multiple of 32 bits.

Note: xdr_inline() might return NULL if there is not enough space in the stream buffer to satisfy the request.

Return Values

The value 1 indicates success; the value 0 indicates an error.

----

xdr_int()

The xdr_int() call translates between C integers and their external representations.

Syntax

```
#include <rpc\rpc.h>

bool_t
xdr_int(xdrs, ip)
XDR *xdrs;
int *ip;
```

Parameters

xdrs
Pointer to the XDR stream.

ip
: Pointer to the integer.

Return Values

The value 1 indicates success; the value 0 indicates an error.

----

xdr_long()

The xdr_long() call translates between C long integers and their external representations.

Syntax

```
#include <rpc\rpc.h>

bool_t
xdr_long(xdrs, lp)
XDR *xdrs;
long *lp;
```

Parameters

xdrs
: Pointer to an XDR stream.

lp
: Pointer to the long integer.

Return Values

The value 1 indicates success; the value 0 indicates an error.

----

xdr_opaque()

The xdr_opaque() call translates between fixed-size opaque data and its external representation.

Syntax

```
#include <rpc\rpc.h>

bool_t
xdr_opaque(xdrs, cp, cnt)
XDR *xdrs;
char *cp;
u_int cnt;
```

Parameters

xdrs
: Pointer to an XDR stream.

cp
: Pointer to the opaque object.

cnt
: Size of the opaque object.

Return Values

The value 1 indicates success; the value 0 indicates an error.

----

xdr_opaque_auth()

The xdr_opaque_auth() call translates between RPC message authentications and their external representations.

Syntax

```
#include <rpc\rpc.h>

bool_t
xdr_opaque_auth(xdrs, ap)
XDR *xdrs;
struct opaque_auth *ap;
```

Parameters

xdrs
Pointer to an XDR stream.

ap
Pointer to the opaque authentication information.

Return Values

The value 1 indicates success; the value 0 indicates an error.

----

xdr_pmap()

The xdr_pmap() call translates an RPC procedure identification, such as is used in calls to Portmapper.

Syntax

```
#include <rpc\rpc.h>

bool_t
xdr_pmap(xdrs, regs)
XDR *xdrs;
struct pmap *regs;
```

Parameters

xdrs
Pointer to an XDR stream.

regs
Pointer to the PORTMAP parameters.

Return Values

The value 1 indicates success; the value 0 indicates an error.

----

xdr_pmaplist()

The xdr_pmaplist() call translates a variable number of RPC procedure identifications, such as those Portmapper creates.

Syntax

```
#include <rpc\rpc.h>

bool_t
xdr_pmaplist(xdrs, rp)
XDR *xdrs;
struct pmaplist **rp;
```

Parameters

xdrs
: Pointer to an XDR stream.

rp
: Pointer to a pointer to the PORTMAP data array.

Return Values

The value 1 indicates success; the value 0 indicates an error.

----

xdr_reference()

The xdr_reference() call provides pointer chasing within structures.

Syntax

```
#include <rpc\rpc.h>

bool_t
xdr_reference(xdrs, pp, size, proc)
XDR *xdrs;
char **pp;
u_int size;
xdrproc_t proc;
```

Parameters

xdrs
: Pointer to an XDR stream.

pp
: Pointer to a pointer.

size
: Size of the target.

proc
: XDR procedure that translates an individual element of the type addressed by the pointer.

Return Values

The value 1 indicates success; the value 0 indicates an error.

----

xdr_rejected_reply()

The xdr_rejected_reply() call translates between rejected RPC reply messages and their external representations.

Syntax

```
#include <rpc\rpc.h>

bool_t
xdr_rejected_reply(xdrs, rr)
XDR *xdrs;
struct rejected_reply *rr;
```

Parameters

xdrs
Pointer to an XDR stream.

rr
Pointer to the rejected reply.

Return Values

The value 1 indicates success; the value 0 indicates an error.

----

xdr_replymsg()

The xdr_replymsg() call translates between RPC reply messages and their external representations.

Syntax

```
#include <rpc\rpc.h>

bool_t
xdr_replymsg(xdrs, rmsg)
XDR *xdrs;
struct rpc_msg *rmsg;
```

Parameters

xdrs
Pointer to an XDR stream.

rmsg
Pointer to the reply message.

Return Values

The value 1 indicates success; the value 0 indicates an error.

----

xdr_setpos()

The xdr_setpos() starts the set-position routine associated with a XDR stream, *xdrs* .

Syntax

```
#include <rpc\rpc.h>

int
xdr_setpos(xdrs, pos)
XDR *xdrs;
```

```
u_int pos;
```

Parameters

xdrs
Pointer to an XDR stream.

pos
Position value obtained from xdr_getpos().

Return Values

The value 1 indicates success; the value 0 indicates an error.

Related Calls

xdr_getpos()

----

xdr_short()

The xdr_short() call translates between C short integers and their external representations.

Syntax

```
#include <rpc\rpc.h>

bool_t
xdr_short(xdrs, sp)
XDR *xdrs;
short *sp;
```

Parameters

xdrs
Pointer to an XDR stream.

sp
Pointer to the short integer.

Return Values

The value 1 indicates success; the value 0 indicates an error.

----

xdr_string()

The xdr_string() call translates between C strings and their external representations.

Syntax

```
#include <rpc\rpc.h>

bool_t
xdr_string(xdrs, sp, maxsize)
XDR *xdrs;
char **sp;
u_int maxsize;
```

Parameters

xdrs
Pointer to an XDR stream.

sp
Pointer to a pointer to the string.

maxsize
Maximum size of the string.

Return Values

The value 1 indicates success; the value 0 indicates an error.

----

xdr_u_int()

The xdr_u_int() call translates between C unsigned integers and their external representations.

Syntax

```
#include <rpc\rpc.h>

bool_t
xdr_u_int(xdrs, up)
XDR *xdrs;
unsigned *up;
```

Parameters

xdrs
Pointer to an XDR stream.

up
Pointer to the unsigned integer.

Return Values

The value 1 indicates success; the value 0 indicates an error.

----

xdr_u_long()

The xdr_u_long() call translates between C unsigned long integers and their external representations.

Syntax

```
#include <rpc\rpc.h>

bool_t
xdr_u_long(xdrs, ulp)
XDR *xdrs;
u_long *ulp;
```

Parameters

xdrs
Pointer to an XDR stream.

ulp

Pointer to the unsigned long integer.

Return Values

The value 1 indicates success; the value 0 indicates an error.

----

xdr_u_short()

The xdr_u_short() call translates between C unsigned short integers and their external representations.

Syntax

```
#include <rpc\rpc.h>

bool_t
xdr_u_short(xdrs, usp)
XDR *xdrs;
u_short *usp;
```

Parameters

xdrs
Pointer to an XDR stream.

usp
Pointer to the unsigned short integer.

Return Values

The value 1 indicates success; the value 0 indicates an error.

----

xdr_union()

The xdr_union() call translates between a discriminated C union and its external representation.

Syntax

```
#include <rpc\rpc.h>

bool_t
xdr_union(xdrs, dscmp, unp, choices, dfault)
XDR *xdrs;
int *dscmp;
char *unp;
struct xdr_discrim *choices;
xdrproc_t dfault;
```

Parameters

xdrs
Pointer to an XDR stream.

dscmp
Pointer to the union's discriminant.

unp
Pointer to the union.

choices

Pointer to an array detailing the XDR procedure to use on each arm of the union.

dfault
Default XDR procedure to use.

Return Values

The value 1 indicates success; the value 0 indicates an error.

----

xdr_vector()

The xdr_vector() call translates between a fixed-length array and its external representation.

Syntax

```
#include <rpc\rpc.h>

bool_t
xdr_vector(xdrs, basep, nelem, elemsize, xdr_elem)
XDR *xdrs;
char *basep;
u_int nelem;
u_int elemsize;
xdrproc_t xdr_elem
```

Parameters

xdrs
Pointer to the XDR stream.

basep
Pointer to the base of the array.

nelem
Element count of the array.

elemsize
Size of each of the array's elements, found by using the sizeof() operator.

xdr_elem
Pointer to the XDR routine that translates an individual array element.

Description

The xdr_vector() call translates between a fixed-length array and its external representation. Unlike variable-length arrays, the storage of fixed-length arrays is static and unfreeable.

Return Values

The value 1 indicates success; the value 0 indicates an error.

----

xdr_void()

The xdr_void() call returns a value of 1.

Syntax

```
#include <rpc\rpc.h>

bool_t
```

```
xdr_void()
```

Description

The xdr_void() call is used like a command that does not require any other XDR functions. You can place this call in the *inproc* or *outproc* parameter of the clnt_call() function when you do not need to move data.

Return Values

The xdr_void() call always returns a value of 1.

Related Calls

- callrpc()
- clnt_broadcast()
- clnt_call()
- clnt_freeres()
- pmap_rmtcall()
- registerrpc()
- svc_freeargs()
- svc_getargs()
- svc_sendreply()

----

xdr_wrapstring()

The xdr_wrapstring() call translates between strings and their external representations.

Syntax

```
#include <rpc\rpc.h>

bool_t
xdr_wrapstring(xdrs, sp)
XDR *xdrs;
char **sp;
```

Parameters

xdrs
: Pointer to an XDR stream.

sp
: Pointer to a pointer to the string.

Description

The xdr_wrapstring() call is the same as calling xdr_string() with a maximum size of MAXUNSIGNED. It is useful because many RPC procedures implicitly start two-parameter XDR routines, and xdr_string() is a three-parameter routine.

Return Values

The value 1 indicates success; the value 0 indicates an error.

----

xdrmem_create()

The xdrmem_create() call initializes the XDR stream pointed to by *xdrs* . Data is written to, or read from, *addr* .

Syntax

```
#include <rpc\rpc.h>

void
xdrmem_create(xdrs, addr, size, op)
XDR *xdrs;
char *addr;
u_int size;
enum xdr_op op;
```

Parameters

xdrs
Pointer to an XDR stream.

addr
Pointer to the memory location.

size
Maximum size of *addr*, in multiples of 4.

op
The direction of the XDR stream (either XDR_ENCODE, XDR_DECODE, or XDR_FREE).

----

xdrrec_create()

The xdrrec_create() call initializes the XDR stream pointed to by *xdrs* .

Syntax

```
#include <rpc\rpc.h>

void
xdrrec_create(xdrs, sendsize, recvsize, handle, readit, writeit)
XDR *xdrs;
u_int sendsize;
u_int recvsize;
char *handle;
int (*readit)();
int (*writeit)();
```

Parameters

xdrs
Pointer to an XDR stream.

sendsize
Size of the send buffer. Specify 0 to choose the default.

recvsize
Size of the receive buffer. Specify 0 to choose the default.

handle
First parameter passed to *readit* () and *writeit* ().

readit ()
Called when a stream's input buffer is empty.

writeit ()
Called when a stream's output buffer is full.

Description

The xdrrec_create() call initializes the XDR stream pointed to by *xdrs* .

Note: The caller must set the *op* field in the xdrs structure.

Attention:

This XDR procedure implements an intermediate record string. Additional bytes in the XDR stream provide record boundary information.

----

xdrrec_endofrecord()

The xdrrec_endofrecord() call marks the data in the output buffer as a completed record.

Syntax

```
#include <rpc\rpc.h>

bool_t
xdrrec_endofrecord(xdrs, sendnow)
XDR *xdrs;
int sendnow;
```

Parameters

xdrs
Pointer to an XDR stream.

sendnow
Specifies nonzero to write out data in the output buffer.

Description

You can start the xdrrec_endofrecord() call only on streams created by xdrrec_create(). Data in the output buffer is marked as a complete record.

Return Values

The value 1 indicates success; the value 0 indicates an error.

----

xdrrec_eof()

The xdrrec_eof() call marks the end of the file, after using the rest of the current record in the XDR stream.

Syntax

```
#include <rpc\rpc.h>

bool_t
xdrrec_eof(xdrs)
XDR *xdrs;
int empty;
```

Parameters

xdrs
Pointer to an XDR stream.

Description

You can start the xdrrec_eof() call only on streams created by xdrrec_create().

Return Values

The value 1 indicates the current record has been consumed; the value 0 indicates continued input on the stream.

----

xdrrec_skiprecord()

The xdrrec_skiprecord() call discards the rest of the XDR stream's current record in the input buffer.

Syntax

```
#include <rpc\rpc.h>

bool_t
xdrrec_skiprecord(xdrs)
XDR *xdrs;
```

Parameters

xdrs
    Pointer to an XDR stream.

Description

You can start the xdrrec_skiprecord() call only on streams created by xdrrec_create(). The XDR implementation is instructed to discard the remaining data in the input buffer.

Return Values

The value 1 indicates success; the value 0 indicates an error.

Related Calls

xdrrec_create()

----

xdrstdio_create()

The xdrstdio_create() call initializes the XDR stream pointed to by *xdrs* . Data is written to or read from the standard I/O stream or file.

Syntax

```
#include <rpc\rpc.h>
#include <stdio.h>

void
xdrstdio_create(xdrs, file, op)
XDR *xdrs;
FILE *file;
enum xdr_op op;
```

Parameters

xdrs
    Pointer to an XDR stream.

file

File name for the input and output stream.

op
: The direction of the XDR stream (either XDR_ENCODE, XDR_DECODE, or XDR_FREE).

----

xprt_register()

The xprt_register() call registers service transport handles with the RPC service package. This routine also modifies the global variable svc_socks[ ].

Syntax

```
#include <rpc\rpc.h>

void
xprt_register(xprt)
SVCXPRT *xprt;
```

Parameters

xprt
: Pointer to the service transport handle.

Related Calls

svc_register()

----

xprt_unregister()

The xprt_unregister() call unregisters the RPC service transport handle.

Syntax

```
#include <rpc\rpc.h>

void
xprt_unregister(xprt)
SVCXPRT *xprt;
```

Parameters

xprt
: Pointer to the service transport handle.

Description

The xprt_unregister() call unregisters an RPC service transport handle. A transport handle should be unregistered with the RPC service package before it is destroyed. This routine also modifies the global variable svc_socks[ ].

File Transfer Protocol API

The following table briefly describes each FTP API call, and identifies where you can find the syntax, parameters, and other appropriate information for these calls.

File Transfer Protocol API Quick Reference

FTP Call	Description
ftpappend()	Appends information to a remote file
ftpcd()	Changes the current working directory on a host
ftpdelete()	Deletes files on a remote host
ftpdir()	Gets a directory in wide format from a host
ftpget()	Gets a file from an FTP server
ftplogoff()	Closes all current connections
ftpls()	Gets directory information in short format from a remote host and writes it to a local file
ftpmkd()	Creates a new directory on a target machine
ftpping()	Resolves a host name and sends a ping to the remote host to determine if the host is responding
ftpproxy()	Transfers a file between two remote servers without sending the file to the local host
ftpput()	Transfers a file to an FTP server
ftpputunique()	Transfers a file to a host and ensures it is created with a unique name
ftppwd()	Stores the string containing the FTP server description of the current working directory on the host to the buffer
ftprename()	Renames a file on a remote host
ftpquote()	Sends a string to the server verbatim
ftpremsize()	returns the size of a file on the remote server.
ftprestart()	The ftprestart() call restarts an aborted transaction from the point of interruption.
ftprmd()	Removes a directory on a target machine
ftpsite()	Executes the site command
ftpsys()	Stores the string containing the FTP server description of the operating system running on the host in a buffer
ftptrcoff()	Closes the trace file, and stops tracing of the command and reply sequences that were sent over the control connection between the local and remote hosts
ftptrcon()	Opens the trace file specified and starts tracing
ftpver()	Stores the string containing the FTP API version
Keep_File_Date()	Maintain the original date/time of files received.

`ping()`	`Sends a ping to the remote host to determine if the host is responding`

Return Values

Most functions return a value of -1 to indicate failure and a value of 0 to indicate success. Two functions do not return 0 and -1 values: ftplogoff(), which is of type void, and ftpping(), which returns an error code rather than storing the return value in *ftperrno* . When the value is -1, the global integer variable *ftperrno* is set to one of the following codes:

Return Code	**Description**
FTPSERVICE	Unknown service.
FTPHOST	Unknown host.
FTPSOCKET	Unable to obtain socket.
FTPCONNECT	Unable to connect to server.
FTPLOGIN	Login failed.
FTPABORT	Transfer aborted.
FTPLOCALFILE	Problem opening the local file.
FTPDATACONN	Problem initializing data connection.
FTPCOMMAND	Command failed.
FTPPROXYTHIRD	Proxy server does not support third party.
FTPNOPRIMARY	No primary connection for proxy transfer.

----

ftpappend()

The ftpappend() call appends information to a remote file.

Syntax

```
#include <ftpapi.h>

int ftpappend(host, userid, passwd, acct,  local,
          remote, transfertype)
char *host;
char *userid;
char *passwd;
char *acct;
char *local;
char *remote;
int transfertype;
```

Parameters

host
Host running the FTP server.

To specify the port number (other than the well-know port) used by the FTP server, code the host name, followed by a blank, then the port number. For example, specify `ftpappend ("server1 1234"`,...)

userid
ID used for logon.

passwd
Password of the user ID.

acct
Account (when needed); can be NULL.

local
: Local file name.

remote
: Remote file name.

transfertype
: Specifies a binary or ASCII transfer. T_ASCII is for ASCII, T_BINARY is for binary.

Return Values

The value 0 indicates success; the value -1 indicates an error. The value of *ftperrno* indicates the specific error. See Return Values for a description of the return codes.

Examples

```
int rc;
rc=ftpappend("conypc","jason","ehgr1",NULL,"abc.doc","new.doc",T_ASCII);
```

The local ASCII file `abc.doc` is appended to the file `new.doc` in the current working directory at the host `conypc`.

----

ftpcd()

The ftpcd() call changes the current working directory on a host.

Syntax

```
#include <ftpapi.h>

int ftpcd(host, userid, passwd, acct, dir)
char *host;
char *userid;
char *passwd;
char *acct,
char *dir;
```

Parameters

host
: Host running the FTP server.

  To specify the port number (other than the well-know port) used by the FTP server, code the host name, followed by a blank, then the port number. For example, specify `ftpcd ("server1 1234"`,...)

userid
: ID used for logon.

passwd
: Password of the user ID.

acct
: Account (when needed); can be NULL.

dir
: New working directory.

Return Values

The value 0 indicates success; the value -1 indicates an error. The value of *ftperrno* indicates the specific error. See Return Values for a description of the return codes.

Examples

```
int rc;
rc=ftpcd("conypc","jason","ehgr1",NULL,"mydir");
```

The current working directory is changed to `mydir` on the host `conypc` using the user ID `jason` and the password `ehgr1`.

----

ftpdelete()

The ftpdelete() call deletes files on a remote host.

Syntax

```
#include <ftpapi.h>

int ftpdelete(host, userid, passwd, acct, name)
char *host;
char *userid;
char *passwd;
char *acct;
char *name;
```

Parameters

host
: Host running the FTP server.

  To specify the port number (other than the well-know port) used by the FTP server, code the host name, followed by a blank, then the port number. For example, specify `ftpdelete ("server1 1234"`,...)

userid
: ID used for logon.

passwd
: Password of the user ID.

acct
: Account (when needed); can be NULL.

name
: File to be deleted.

Return Values

The value 0 indicates success; the value -1 indicates an error. The value of *ftperrno* indicates the specific error. See Return Values for a description of the return codes.

Examples

```
int rc;
rc=ftpdelete("conypc","jason","ehgr1",NULL,"abc.1");
```

The file `abc.1` is deleted on the host `conypc` using the user ID `jason` and the password `ehgr1`.

----

ftpdir()

The ftpdir() call gets a directory in wide format from a host.

Syntax

```
#include <ftpapi.h>
```

```
int ftpdir(host, userid, passwd, acct, local, pattern,)
char *host;
char *userid;
char *passwd;
char *acct;
char *local;
char *pattern;
```

Parameters

host
: Host running the FTP server.

  To specify the port number (other than the well-know port) used by the FTP server, code the host name, followed by a blank, then the port number. For example, specify `ftpdir ("server1 1234",...)`

userid
: ID used for logon.

passwd
: Password of the user ID.

acct
: Account (when needed); can be NULL.

local
: Local file name.

pattern
: The file name or pattern of the files to be listed on the foreign host. Patterns are any combination of ASCII characters. The following two characters have special meaning:

*	Shows that any character or group of characters can occupy that position in the pattern.
?	Shows that any single character can occupy that position in the pattern.

Return Values

The value 0 indicates success; the value -1 indicates an error. The value of *ftperrno* indicates the specific error. See Return Values for a description of the return codes.

Examples

```
int rc;
rc=ftpdir("conypc","jason","ehgr1",NULL,"conypc.dir","*.c");
```

ftpdir() gets a directory of `*.c` files in wide format, and stores the directory in a local file, `conypc.dir`.

----

ftpget()

The ftpget() call gets a file from an FTP server.

Syntax

```
#include <ftpapi.h>

int ftpget(host, userid, passwd, acct,  local, remote,
       mode, transfertype)
char *host;
char *userid;
char *passwd;
char *acct;
char *local;
char *remote;
char *mode;
```

```
int transfertype;
```

Parameters

host
: Host running the FTP server.

  To specify the port number (other than the well-know port) used by the FTP server, code the host name, followed by a blank, then the port number. For example, specify `ftpget ("server1 1234"`,...)

userid
: ID used for logon.

passwd
: Password of the user ID.

acct
: Account (when needed); can be NULL.

local
: Local file name.

remote
: Remote file name.

mode
: Either *w* for write or *a* for append.

transfertype
: Specifies a binary or ASCII transfer. T_ASCII is for ASCII, T_BINARY is for binary.

Return Values

The value 0 indicates success; the value -1 indicates an error. The value of *ftperrno* indicates the specific error. See Return Values for a description of the return codes.

Examples

```
int rc;
rc=ftpget("conypc","jason","ehgr1",NULL,"new.doc","abc.doc","w",T_ASCII);
```

The system copies the ASCII file `abc.doc` on the host `conypc` into the local current working directory as the file `new.doc`. If the file `new.doc` already exists in the local current working directory, the contents of the file `abc.doc` overwrite the file `new.doc`.

----

ftplogoff()

The ftplogoff() call closes all current connections. An application must call this before terminating.

Syntax

```
#include <ftpapi.h>

void ftplogoff()
```

----

ftpls()

The ftpls() call gets directory information in short format from a remote host and writes it to a local file.

Syntax

```
#include <ftpapi.h>

int ftpls(host, userid, passwd, acct, local, pattern)
char *host;
char *userid;
char *passwd;
char *acct;
char *local;
char *pattern;
```

Parameters

host
: Host running the FTP server.

  To specify the port number (other than the well-know port) used by the FTP server, code the host name, followed by a blank, then the port number. For example, specify `ftpls ("server1 1234"`,...)

userid
: ID used for logon.

passwd
: Password of the user ID.

acct
: Account (when needed); can be NULL.

local
: Local file into which the information is placed.

pattern
: The file name or pattern of the files to be listed on the foreign host. Patterns are any combination of ASCII characters. The following two characters have special meaning:

***	Shows that any character or group of characters can occupy that position in the pattern.
?	Shows that any single character can occupy that position in the pattern.

Return Values

The value 0 indicates success; the value -1 indicates an error. The value of *ftperrno* indicates the specific error. See Return Values for a description of the return codes.

Examples

```
int rc;
rc=ftpls("conypc","jason","ehgr1",NULL,"conypc.dir","*.c");
```

ftpls() gets a directory of `*.c` files in short format and stores the names in the local file `conypc.dir`.

----

ftpmkd()

The ftpmkd() call creates a new directory on a target machine.

Syntax

```
#include <ftpapi.h>

int ftpmkd(host, userid, passwd, acct, dir)
char *host;
char *userid;
char *passwd;
char *acct;
char *dir;
```

Parameters

host
: Host running the FTP server

  To specify the port number (other than the well-know port) used by the FTP server, code the host name, followed by a blank, then the port number. For example, specify `ftpmkd ("server1 1234"`,...)

userid
: ID used for logon.

passwd
: Password of the user ID.

acct
: Account (when needed); can be NULL.

dir
: Directory to be created.

Return Values

The value 0 indicates success; the value -1 indicates an error. The value of *ftperrno* indicates the specific error. See Return Values for a description of the return codes.

Examples

```
int rc;
rc=ftpmkd("conypc","jason","ehgr1",NULL,"mydir");
```

The directory `mydir` is created on the host `conypc`, using the user ID `jason` and the password `ehgr1`.

----

ftpping()

The ftpping() call resolves a host name and sends a ping to the remote host to determine if the host is responding.

Syntax

```
#include <ftpapi.h>

int ftpping(host, len, addr)
char *host;
int len;
unsigned long *addr;
```

Parameters

host
: Host running the FTP server.

len
: Length of the ping packets.

addr
: Buffer in which to return the host internet address.

Description

The ftpping() call tries to resolve the host name through a name server. If the name server is not present, ftpping() searches the TCPIP\ETC\HOSTS file for a matching host name. Unlike the ping() call, ftpping() could take several seconds because it must resolve the host name before it sends a ping. For this reason, use ftpping() only in the first try to determine if the host is responding. The ftpping() call sets the *addr* parameter to the internet address of the host. After the first try, use this address value to call ping.

If the ftpping() return value is positive, the return value is the number of milliseconds it took for the echo to return. If the return value is negative, it contains an error code. The parameter *len* specifies the length of the ping packet(s).

Return Values

The following are ftpping() call return codes and their corresponding descriptions:

Return Code	**Description**
PINGREPLY	Host does not reply
PINGSOCKET	Unable to obtain socket
PINGPROTO	Unknown protocol ICMP
PINGSEND	Send failed
PINGRECV	Recv() failed
PINGHOST	Unknown host

Examples

```
int              rc;
unsigned long    addr;

rc = ftpping("conypc", 256, &addr);
```

The ftpping() call sends a 256-byte ping packet to the host `conypc`.

----

ftpproxy()

The ftpproxy() call transfers a file between two remote servers without sending the file to the local host.

Syntax

```
#include <ftpapi.h>

int ftpproxy(host1, userid1, passwd1, acct1, host2, userid2,
         passwd2, acct2, fn1, fn2, transfertype)
char *host1;
char *userid1;
char *passwd1;
char *acct1;
char *host2;
char *userid2;
char *passwd2;
char *acct2;
char *fn1;
char *fn2;
int transfertype;
```

Parameters

host1
Target host running the FTP server.

To specify the port number (other than the well-know port) used by the FTP server, code the host name, followed by a blank, then the port number. For example, specify `ftpproxy ("server1 1234",`...)

userid1
ID used for logon on host 1.

passwd1
Password of the user ID on host 1.

acct1
Account for host 1 (when needed); can be NULL.

host2
: Source host running the FTP server.

  To specify the port number (other than the well-know port) used by the FTP server, code the host name, followed by a blank, then the port number. For example, specify `ftpproxy (host1,..."server2 1234"`,...)

userid2
: ID used for logon on host 2.

passwd2
: Password of the user ID on host 2.

acct2
: Account for host 2 (when needed); can be NULL.

fn1
: File to be written on host 1.

fn2
: File to be copied from host 2.

transfertype
: Specifies a binary or ASCII transfer. T_ASCII is for ASCII, T_BINARY is for binary.

Description

The ftpproxy() call copies a file on a specified source host directly to a specified target host, without involving the requesting host in the file transfer. This call is functionally the same as the FTP client subcommand *proxy put* .

Notes:

1. For ftpproxy() to complete successfully, both the source and the target hosts must be running the FTP servers. In addition, ftpproxy() does not support connections through a firewall.

2. You can specify port number for either *host1* or *host2* , or for both.

Return Values

The value 0 indicates success; the value -1 indicates an error. The value of *ftperrno* indicates the specific error. See Return Values for a description of the return codes.

Examples

```
int rc;
rc=ftpproxy("pc1","oleg","erst",NULL,   /* target host information*/
            "pc2","yan", "dssa1", NULL, /* source host information*/
            "\tmp\newdoc.1",            /* target file name */
            "\tmp\doc.1",               /* source file name */
            T_ASCII);                   /* ASCII transfer */
```

The ASCII file `\tmp\doc.1` on the host `pc2` is copied to host `pc1` as the file `\tmp\newdoc.1`.

----

ftpput()

The ftpput() call transfers a file to an FTP server.

Syntax

```
#include <ftpapi.h>

int ftpput(host, userid, passwd, acct, local, remote,
       transfertype)
char *host;
char *userid;
char *passwd;
char *acct;
char *local;
char *remote;
int transfertype;
```

Parameters

host
: Host running the FTP server.

  To specify the port number (other than the well-know port) used by the FTP server, code the host name, followed by a blank, then the port number. For example, specify `ftpput ("server1 1234",...)`

userid
: ID used for logon.

passwd
: Password of the user ID.

acct
: Account (when needed); can be NULL.

local
: Local file name.

remote
: Remote file name.

transfertype
: Specifies a binary or ASCII transfer. T_ASCII is for ASCII, T_BINARY is for binary.

Return Values

The value 0 indicates success; the value -1 indicates an error. The value of *ftperrno* indicates the specific error. See Return Values for a description of the return codes.

Examples

```
int rc;
rc=ftpput("conypc","jason","ehgr1",NULL,"abc.doc","new.doc",T_ASCII);
```

The system copies the ASCII file `abc.doc` on the local current working directory to the current working directory of the host `conypc` as file `new.doc`. If the file `new.doc` already exists, the contents of the file `abc.doc` overwrite the file `new.doc`.

----

ftpputunique()

The ftpputunique() call transfers a file to a host and ensures it is created with a unique name.

Syntax

```
#include <ftpapi.h>

int ftpputunique(host, userid, passwd, acct, local, remote,
                 transfertype)
char *host;
char *userid;
char *passwd;
char *acct;
char *local;
char *remote;
int transfertype;
```

Parameters

host

Host running the FTP server.

To specify the port number (other than the well-know port) used by the FTP server, code the host name, followed by a blank, then the port number. For example, specify `ftpputunique ("server1 1234"`,...)

userid
: ID used for logon.

passwd
: Password of the user ID.

acct
: Account (when needed); can be NULL.

local
: Local file name.

remote
: Remote file name.

transfertype
: Specifies a binary or ASCII transfer. T_ASCII is for ASCII, T_BINARY is for binary.

Description

The ftpputunique() call copies a local file to a file on a specified host. It guarantees that the new file will have a unique name and that the new file will not overwrite a file with the same name. If the file already exists on the host, a new and unique file name is created and used as the target of the file transfer.

Return Values

The value 0 indicates success; the value -1 indicates an error. The value of *ftperrno* indicates the specific error. See Return Values for a description of the return codes.

Examples

```
int rc;
rc=ftpputunique(
       "conypc","jason","ehgr1",NULL,"abc.doc", "new.doc",T_ASCII);
```

The ASCII file `abc.doc` is copied to the current working directory of the host `conypc` as file `new.doc`, unless the file `new.doc` already exists. If the file `new.doc` already exists, the file `new.doc` is given a new name unique within the current working directory on the host `conypc`. The name of the new file is displayed upon successful completion of the file transfer.

----

ftppwd()

The ftppwd() call stores the string containing the FTP server description of the current working directory on the host to the buffer.

Syntax

```
#include <ftpapi.h>

int ftppwd(host, userid, passwd, acct, buf, buflen)
char *host;
char *userid;
char *passwd;
char *acct,
char *buf;
crt *buflen;
```

Parameters

host
: Host running the FTP server.

To specify the port number (other than the well-know port) used by the FTP server, code the host name, followed by a blank, then the port number. For example, specify `ftppwd ("server1 1234"`,...)

userid
: ID used for logon.

passwd
: Password of the user ID.

acct
: Account (when needed); can be NULL.

buf
: Buffer to store the string returned by the FTP server.

buflen
: Length of *buf* .

Description

The ftppwd() call stores the string containing the FTP server description of the current working directory on the host to the buffer *buf* . The string describing the current working directory is truncated to fit *buf* if it is longer than *buflen* . The returned string is always null-terminated.

Return Values

The value 0 indicates success; the value -1 indicates an error. The value of *ftperrno* indicates the specific error. See Return Values for a description of the return codes.

Examples

```
int rc;
rc=ftppwd("conypc","jason","ehgr1","dirbuf", sizeof dirbuf);
```

After the ftppwd() call, the buffer `dirbuf` contains the following:

```
"C:\" is current directory.
```

The server reply describing the current working directory on host `conypc` using user ID `jason` with password `eghr1` is stored to `dirbuf`.

----

ftpquote()

The ftpquote() call sends a string to the server verbatim.

Syntax

```
#include <ftpapi.h>

int ftpquote(host, userid, passwd, acct, quotestr)
char *host;
char *userid;
char *passwd;
char *acct;
char *quotestr;
```

Parameters

host
: Host running the FTP server.

To specify the port number (other than the well-know port) used by the FTP server, code the host name, followed by a blank, then the port number. For example, specify `ftpquote ("server1 1234"`,...)

userid
: ID used for logon.

passwd
: Password of the user ID.

acct
: Account (when needed); can be NULL.

quotestr
: Quote string to be passed to the FTP server verbatim.

Return Values

The value 0 indicates success; the value -1 indicates an error. The value of *ftperrno* indicates the specific error. See Return Values for a description of the return codes.

Examples

```
int rc;
rc=ftpquote("conypc","jason","ehgr1",NULL,"site idle 2000");
```

The idle is set to time out in 2000 seconds. Your server might not support that amount of idle time.

----

ftpremsize()

The ftpremsize() call returns the size of a file on the remote server.

Syntax

```
#include <ftpapi.h>

int ftpget(host, userid, passwd, acct,  local, remote,
        mode, transfertype)
char *host;
char *userid;
char *passwd;
char *acct;
char *local;
char *remote;
char *mode;
int transfertype;
```

Parameters

host
: Host running the FTP server.

  To specify the port number (other than the well-know port) used by the FTP server, code the host name, followed by a blank, then the port number. For example, specify `ftpget ("server1 1234"`,...)

userid
: ID used for logon.

passwd
: Password of the user ID.

acct
: Account (when needed); can be NULL.

local
: Local file name.

remote
: Remote file name.

mode

Either *w* for write or *a* for append.

transfertype
Specifies a binary or ASCII transfer. T_ASCII is for ASCII, T_BINARY is for binary.

Return Values

Value greater than zero indicates success which is the size of the remote file, the value -1 indicates error.

----

ftprename()

The ftprename() call renames a file on a remote host.

Syntax

```
#include <ftpapi.h>

int ftprename(host, userid, passwd, acct, namefrom, nameto)
char *host;
char *userid;
char *passwd;
char *acct;
char *namefrom;
char *nameto;
```

Parameters

host
Host running the FTP server.

To specify the port number (other than the well-know port) used by the FTP server, code the host name, followed by a blank, then the port number. For example, specify `ftprename ("server1 1234"`,...)

userid
ID used for logon.

passwd
Password of the user ID.

acct
Account (when needed); can be NULL.

namefrom
Original file name.

nameto
New file name.

Return Values

The value 0 indicates success; the value -1 indicates an error. The value of *ftperrno* indicates the specific error. See Return Values for a description of the return codes.

Examples

```
int rc;
rc=ftprename("conypc","jason","ehgr1",NULL,"abc.1","cd.fg");
```

The file `abc.1` is renamed to `cd.fg` on host `conypc`, using user ID `jason`, with password `ehgr1`.

----

ftprestart()

The ftprestart() call restarts an aborted transaction from the point of interruption.

Syntax

```
#include <ftpapi.h>

int ftprestart(host, userid, passwd, acct,  local, remote,
       mode, transfertype,
       rest)
char *host;
char *userid;
char *passwd;
char *acct;
char *local;
char *remote;
char *mode;
int transfertype;
int rest;
```

Parameters

host
: Host running the FTP server.

  To specify the port number (other than the well-know port) used by the FTP server, code the host name, followed by a blank, then the port number. For example, specify `ftpget ("server1 1234"`,...)

userid
: ID used for logon.

passwd
: Password of the user ID.

acct
: Account (when needed); can be NULL.

local
: Local file name.

remote
: Remote file name.

mode
: Either *w* for write or *a* for append.

transfertype
: Specifies a binary or ASCII transfer. T_ASCII is for ASCII, T_BINARY is for binary.

rest
: Flag to indicate whether, it is the restart of a GET trransaction or a PUT transaction.

Return Values

Value greater than zero indicates success, the value -1 indicates error.

----

ftprmd()

The ftprmd() call removes a directory on a target machine.

Syntax

```
#include <ftpapi.h>
```

```
int ftprmd(host, userid, passwd, acct, dir)
char *host;
char *userid;
char *passwd;
char *acct;
char *dir;
```

Parameters

host
Host running the FTP server.

To specify the port number (other than the well-know port) used by the FTP server, code the host name, followed by a blank, then the port number. For example, specify `ftprmd ("server1 1234"`,...)

userid
ID used for logon.

passwd
Password of the user ID.

acct
Account (when needed); can be NULL.

dir
Directory to be removed.

Return Values

The value 0 indicates success; the value -1 indicates an error. The value of *ftperrno* indicates the specific error. See Return Values for a description of the return codes.

Examples

```
int rc;
rc=ftprmd("conypc","jason","ehgr1",NULL,"mydir");
```

The directory, `mydir`, is removed on the host, `conypc`, using the user ID, `jason`, and the password, `ehgr1`.

----

ftpsite()

The ftpsite() call executes the **site** command. (For more information about the **site** command, see the *TCP/IP Command Reference* .)

Note: ftpsite() does not support connections through a firewall.

Syntax

```
#include <ftpapi.h>

int ftpsite(host, userid, passwd, acct, sitestr)
char *host;
char *userid;
char *passwd;
char *acct;
char *sitestr;
```

Parameters

host
Host running the FTP server.

To specify the port number (other than the well-know port) used by the FTP server, code the host name, followed by a blank, then the port number. For example, specify `ftpsite ("server1 1234"`,...)

userid
: ID used for logon.

passwd
: Password of the user ID.

acct
: Account (when needed); can be NULL.

sitestr
: Site string to be executed.

Return Values

The value 0 indicates success; the value -1 indicates an error. The value of *ftperrno* indicates the specific error. See Return Values for a description of the return codes.

Examples

```
int rc;
rc=ftpsite("conypc","jason","ehgr1",NULL,"idle 2000");
```

The idle is set to time out in 2000 seconds. Your server might not support that amount of idle time.

---

ftpsys()

The ftpsys() call stores the string containing the FTP server description of the operating system running on the host in a buffer.

Syntax

```
#include <ftpapi.h>

int ftpsys(host, userid, passwd, acct, buf, buflen)
char *host;
char *userid;
char *passwd;
char *acct,
char *buf;
int *buflen;
```

Parameters

host
: Host running the FTP server.

  To specify the port number (other than the well-know port) used by the FTP server, code the host name, followed by a blank, then the port number. For example, specify `ftpsys ("server1 1234"`,...)

userid
: ID used for logon.

passwd
: Password of the user ID.

acct
: Account (when needed); can be NULL.

buf
: Buffer to store the string returned by the FTP server.

buflen
: Length of *buf* .

Description

The ftpsys() call stores the string containing the FTP server description of the operating system running on the host in the buffer *buf*. The string describing the operating system of the host is truncated to fit *buf* if it is longer than *buflen*. The returned string is always null-terminated.

Return Values

The value 0 indicates success; the value -1 indicates an error. The value of *ftperrno* indicates the specific error. See Return Values for a description of the return codes.

Examples

```
int rc;
rc=ftpsys("ralvmm","jason","ehgr1",hostsysbuf, sizeof hostsysbuf);
```

After the ftpsys() call the buffer `hostsysbuf` contains the following:

```
VM is the operating system of this server.
```

The FTP server reply describing the operating system of host `ralvmm` using user ID `jason` with password `eghr1` is stored to `hostsysbuf`.

----

ftptrcoff()

The ftptrcoff() closes the trace file, and stops tracing of the command and reply sequences that were sent over the control connection between the local and remote hosts.

Syntax

```
#include <ftpapi.h>

int ftptrcoff(void)
```

Return Values

The ftptrcoff() always return a value of 0.

Examples

```
int rc;
rc = ftptrcoff();
```

----

ftptrcon()

The ftptrcon() call opens the trace file specified and starts tracing.

Syntax

```
#include <ftpapi.h>

int ftptrcon(fileSpec, mode)
char *fileSpec;
int mode;
```

Parameters

fileSpec

Identifies the name of the trace file.

mode
Specifies the trace mode as overwrite or append. Use M_OVERLAY for trace data which overwrites previous information. Use M_APPEND for trace data which appends to previous information.

Description

The ftptrcon() call opens the trace file specified and starts tracing of the command and reply sequences sent over the control connection between the local and remote hosts. The trace file can be written over or appended to.

No notification is provided if writing of trace data fails.

Telnet command and reply sequences are not traced nor are command and reply sequences between the local host and a proxy host.

Return Values

There are three possible return values for ftptrcon():

- 0 when successful
- TRCMODE indicates the value set into mode was not valid
- TRCOPEN indicates the trace file could not be opened

Examples

To write the trace data into a file named api.trc in the C:\WORK directory, use :

```
int rc;
rc = ftptrcon("c\\work\\api.trc", M_OVERLAY);
```

If the file already existed, the new trace data overwrites the previous trace data (overlay mode).

----

ftpver()

The ftpver() call stores the string containing the FTP API version.

Syntax

```
#include <ftpapi.h>

int ftpver(buf, buflen)
char *buf;
int buflen;
```

Parameters

buf
Identifies the buffer to store the version string.

buflen
Specifies the length of the buffer.

Description

The ftpver() call stores the string containing the FTP API version. The string is truncated to fit into the buffer if it is longer than the buffer length. The returned string is always null-terminated.

Return Values

The value of 0 when successful. The value of -1 when the complete version string could not be copied because the buffer length was too small.

Examples

```
int rc;
```

```
rc = ftpver(verBuf, bufLen);
```

After the ftpver() call, the buffer contains the version number.

Keep_File_Date()

Maintains the original date/time of files received. This utilises the MDTM command which returns gmt YYYYMMDDHHMMSS format.

Syntax

```
#include <ftpapi.h>

BOOL Keep_File_Date( localfile, remotefile,)
char *localfile;
char *remotefile;
```

Parameters

localfile
Name of the file on the local machine to which the date/time of the remotefile has to be assigned.

remotefile
Name of the file on the server, for which, the date is to be maintained.

Return Values

The value TRUE indicates Success, Value FALSE indicates failure implying that the original date/time could not be maintained.

----

ping()

The ping() call sends a ping to the remote host to determine if the host is responding.

Syntax

```
#include <ftpapi.h>

int ping(addr, len)
unsigned long addr;
int len;
```

Parameters

addr
Internet address of the host in network byte order.

len
Length of the ping packets.

Description

The ping() call sends a ping to the host with ICMP Echo Request. The ping() call is useful to determine whether the host is alive before attempting FTP transfers, because time-out on regular connections is more than a minute. The ping() call returns within 3 seconds, at most, if the host is not responding.

Return Values

If the return value is positive, the return value is the number of milliseconds it took for the echo to return. If the return value is negative, it contains an error code.

The following are ping() call return codes and their corresponding descriptions:

Return Code	**Description**
PINGREPLY	Host does not reply
PINGSOCKET	Unable to obtain socket
PINGPROTO	Unknown protocol ICMP
PINGSEND	Send failed
PINGRECV	Recv() failed

Examples

```
#include <stdio.h>
#include <netdb.h>
#include <ftpapi.h>

struct hostent *hp;     /* Pointer to host info */

main(int argc, char *argv[], char *envp[])
{
   int i;
   unsigned long addr;

   if (argc!=2) {
     printf("Usage: p <host>\n");
     exit(1);
   }

   hp = gethostbyname(argv[1]);



if (hp) {
        memcpy( (char *)&addr, hp->h_addr, hp->h_length);
        i = ping(addr,256);
        printf("ping reply in %d milliseconds\n",i);
} else {
        printf("unknown host\n");
        exit(2);
       }
       ftplogoff(); /* close all connections */
}
```

Resource ReSerVation Protocol API

This table briefly describes each RSVP API call, and provides links to the syntax, parameters, and other appropriate information for these calls:

RSVP API Quick Reference

RSVP Call	Description
rapi_dispatch()	Dispatches an RSVP event
rapi_fmt_adspec()	Formats adspec information for printng
rapi_fmt_filtspec()	Formats filterspec information for printing
rapi_fmt_flowspec()	Formats flowspec information for printing
rapi_fmt_tspec()	Formats tspec information for printing
rapi_getfd()	Gets the alert socket for a session
rapi_release()	Ends an RSVP session

`rapi_reserve()`	`Makes a reservation to be a receiver`
`rapi_sender()`	`Specifies parameters to become a sender`
`rapi_session()`	`Starts an RSVP session`
`rapi_version()`	`Gets the RSVP version`
`user_rapi_callback()`	`Initializes the user callback function`

Return Values and Definitions

These return values have these meanings for RSVP calls:

RAPI_ERR_OK	No error.
RAPI_ERR_INVAL	Parameter not valid.
RAPI_ERR_MAXSESS	Too many sessions.
RAPI_ERR_BADSID	Session identifier out of valid range.
RAPI_ERR_N_FFS	Wrong n_filter or n_flow for this style.
RAPI_ERR_BADSTYLE	Illegal reservation style.
RAPI_ERR_SYSCALL	System error. See errno.
RAPI_ERR_OVERFLOW	Parameter list overflow.
RAPI_ERR_MEMFULL	Not enough memory.
RAPI_ERR_NORSVP	Daemon does not respond or does not exist.
RAPI_ERR_OBJTYPE	Object type error.
RAPI_ERR_OBJLEN	Object length error.
RAPI_ERR_NOTSPEC	No sender tspec in rapi_sender.
RAPI_ERR_INTSERV	Intserv format error.
RAPI_ERR_BADSEND	Sender interface does not exist.
RAPI_ERR_BADRECV	Receiver interface does not exist.
RAPI_ERR_UNSUPPORTED	Unsupported return code.
RAPI_ERR_UNKNOWN	Unknown return code.

----

rapi_dispatch()

The rapi_dispatch() call dispatches an RSVP event.

Syntax

```
#include <rsvprapi.h>

int rapi_dispatch(void);
```

Parameters: None.

Description

The rapi_getfd() call returns a socket for the session. The socket may be used with the select() call as a read socket. When select() indicates that there is some data to read, call rapi_dispatch() call to read and process the data from the socket. Typically rapi_dispatch() will call user_rapi_callback() to provide the program with information. The user program should only process data from the socket by calling rapi_dispatch().

Return Values and Descriptions

These return values indicate the specific errors for rapi_dispatch():

RAPI_ERR_INVAL	Conversion from RSVP daemon structures to API structures encountered data values that are not valid or not supported.
RAPI_ERR_MEMFULL	Could not allocate memory.
RAPI_ERR_NORSVP	Could not connect to the RSVP daemon. It may not be running.

Related Calls

rapi_getfd()
select()
user_rapi_callback()

----

rapi_fmt_adspec()

The rapi_fmt_adspec() call formats adspec information as a printable string.

Syntax

```
#include <rsvprapi.h>

void rapi_fmt_adspec(
       rapi_adspec_t *padspec,
       char          *buffer,
       int            length);
```

Parameters

padspec	Pointer to an adspec.
buffer	Pointer to a buffer where the information will be put for printing.
length	The number of characters the buffer can hold.

Description

The rapi_fmt_adspec() call formats the information in an adspec into the buffer, in a form suitable for printing. The output information is truncated if the buffer is too small.

Return Values

None.

Related Calls

rapi_fmt_filtspec()
rapi_fmt_flowspec()
rapi_fmt_tspec()

----

rapi_fmt_filtspec()

The rapi_fmt_filtspec() call formats filterspec information as a printable string.

Syntax

```
#include <rsvprapi.h>

void rapi_fmt_filtspec (
        rapi_filter_t *pfilter,
        char          *buffer
        int            length);
```

Parameters

pfilter

Pointer to a filterspec.

buffer

Pointer to a buffer where the information will be put for printing.

length

The number of characters the buffer can hold.

Description

The rapi_fmt_filtspec() call formats the information in a filterspec into the buffer, in a form suitable for printing. The output information is truncated if the buffer is too small.

Return Values

None.

Related Calls

rapi_fmt_adspec()
rapi_fmt_flowspec()
rapi_fmt_tspec()

----

rapi_fmt_flowspec()

The rapi_fmt_flowspec() call formats flowspec information as a printable string.

Syntax

```
#include <rsvprapi.h>

void rapi_fmt_flowspec(
       rapi_flowspec_t *pflowspec,
       char            *buffer,
       int              length);
```

Parameters

pflowspec

Pointer to a flowspec.

buffer

Pointer to a buffer where the information will be put for printing.

length

The number of characters the buffer can hold.

Description

The rapi_fmt_flowspec() call formats the information in a flowspec into the buffer, in a form suitable for printing. The output information is truncated if the buffer is too small.

Return Values

None.

Related Calls

rapi_fmt_adspec()
rapi_fmt_filtspec()
rapi_fmt_tspec()

----

rapi_fmt_tspec()

The rapi_fmt_tspec() call formats tspec information as a printable string.

Syntax

```
#include <rsvprapi.h>

void rapi_fmt_tspec(
       rapi_tspec_t *ptspec,
       char         *buffer,
       int           length);
```

Parameters

ptspec

Pointer to a tspec.

buffer

Pointer to a buffer where the information will be put for printing.

length

The number of characters the buffer can hold.

Description

The rapi_fmt_tspec() call formats the information in a tspec into the buffer, in a form suitable for printing. The output information is truncated if the buffer is too small.

Return Values

None.

Related Calls

rapi_fmt_adspec()
rapi_fmt_filtspec()
rapi_fmt_flowspec()

----

rapi_getfd()

The rapi_getfd() call obtains the alert socket for a session.

Syntax

```
#include <rsvprapi.h>

int  rapi_getfd(
       rapi_sid_t sid);
```

Parameters

sid	Session identifier.

Description

The rapi_getfd() call formats the information in a tspec into the buffer, in a form suitable for printing. The output information is truncated if the buffer is too small.

Return Values

A socket number is returned, or -1. The value -1 indicates an invalid session ID was used as an argument.

Related Calls

rapi_dispatch()
rapi_session()
select()
user_rapi_callback()

----

rapi_release()

The rapi_release() call ends an RSVP session.

Syntax

```
#include <rsvprapi.h>

int  rapi_release(
       rapi_sid_t sid);
```

Parameters

sid	Session identifier of an open session.

Description

The rapi_release() call closes an open session. The session ID will no longer be valid.

Return Values and Descriptions

RAPI_ERR_BADSID	The session identifier is invalid, or the session is not open
RAPI_ERR_NORSVP	The API could not communicate with the RSVP daemon.
RAPI_ERR_OK	The session was closed successfully.

Related Calls

rapi_session()

----

rapi_reserve()

The rapi_reserve() call makes a reservation to be a receiver.

Syntax

```
#include <rsvprapi.h>

int  rapi_reserve(
       rapi_sid_t         sid,
       int                flag,
       struct sockaddr   *phost,
       rapi_sytleid_t     style,
       rapi_stylex_t     *pstyle,
       rapi_policy_t     *ppolicy,
       int                numFilters,
       rapi_filter_t     *pfilter,
       int                numFlows,
       rapi_flowspec_t   *pflow);
```

Parameters

sid	Session identifier.
flag	Only the optional RAPI_REQ_CONFIRM flag can be used, or a zero. Using this flag requests a RAPI_RESV_CONFIRM event be provided to the callback function when the reservation is complete. This indicates merely a high probability that the reservation was completed, not that it is certain.
phost	Receive host address and port, or NULL. For this implementation, this is a pointer to a sockaddr_in structure. If the address is INADDR_ANY, or if *phost* is NULL, the default interface will be used.
style	The reservation style may be RAPI_RSTYLE_WILDCARD, RAPI_RSTYLE_FIXED, or RAPI_RSTYLE_SE.
pstyle	Style extension (not supported).
ppolicy	Receiver policy (not supported).
numFilters	Number of filterspecs pointed to by pfilter.
pfilter	An array of filterspecs.
numFlows	The number of flowspecs pointed to by pflow. If 0, the current reservation for the session is removed, if there is one.
pflow	An array of flowspecs.

Description

The rapi_reserve() call establishes the user of the session as a receiver. It specifies a reservation style and a filterspec array and flowspec array. If there is a previous reservation still in effect, and the rapi_reserve() call specifies a different one, the new reservation replaces the previous one. If the number of flowspecs is 0, the current reservation is deleted but no new reservation is made.

After successfully calling rapi_reserve(), the application callback function can be called with RAPI_RESV_ERROR or RAPI_RESV_CONFIRM events.

There are three reservation styles:

- Fixed filter (RAPI_RSTYLE_FIXED) specifies one or more senders in the array of filterspecs, and an equal number of flowspecs. The i-th flowspec is associated with the i-th filterspec.
- Shared explicit filter (RAPI_RSTYLE_SE) specifies one or more senders in the array of filterspecs, and one flowspec. All the senders are expected to match the flowspec.
- Wildcard filter (RAPI_RSTYLE_WILDCARD) specifies a single flowspec, and either no filterspec, or a single filterspec with appropriate wildcard(s). If no sender is specified with a filterspec, any sender that matches the flowspec is a valid sender.

Filterspecs have two formats:

- RAPI_FILTERFORM_1 allows for a wildcard specification of allowable senders. This format is not supported in this implementation of the RSVP API.
- RAPI_FILTERFORM_BASE has a sockaddr_in structure that specifies a sender IP address and port.

Flowspecs have the following formats:

- RAPI_FORMAT_IS_CL specifies a controlled load flowspec.
- RAPI_FORMAT_IS_GUAR specifies a guarenteed flowspec.

The flowspec data fields relate to rapi_reserve().

Return Values and Descriptions

RAPI_ERR_BADSID	The session identifier is not valid, or the session is not open.
RAPI_ERR_INVAL	The argument is not valid.
RAPI_ERR_OK	The daemon accepted the reservation. Asynchronous callbacks may report further status.

Related Calls

rapi_sender()

----

rapi_sender()

The rapi_sender() call provides information required to become a sender.

Syntax

```
       #include <rsvprapi.h>

int  rapi_sender(
       rapi_sid_t          sid,
       int                 flags,
       struct sockaddr    *plocal,
       rapi_filter_t      *pfilter,
       rapi_tspec_t       *ptraffic,
       rapi_adspec_t      *padvert,
       rapi_policy_t      *ppolicy,
       int                 ttl);
```

Parameters

sid

Session identifier.

flags

No flags are supported; specify 0.

plocal

Local host (src addr, port). This argument points to a structure that specifies the interface that will be used to send the data. For this implementation, this should be a sockaddr_in structure. If the IP source address is INADDR_ANY, the default IP address of the host will be used. If plocal is NULL, the program is withdrawing its registration as a sender for the session and the other arguments will be ignored.

pfilter

Sender template. This parameter is not supported. Specify NULL.

ptraffic

Sender tspec. This parameter is a pointer to the traffic specification for the data flow that this sender will send.

padvent

Sender adspec. This parameter is not supported; specify NULL.

ppolicy

Sender policy data. This parameter is not supported; specify NULL.

ttl

Time to live of the multicast data. If sending data to a multicast group, specify the TTL used to send to that group, as specified with setsockopt() option IP_MULTICAST_TTL.

Description

The rapi_sender() call establishes the program as a sender for the specified session. After checking the arguments for validity, this call sends an appropriate message to the destination.

After successfully calling rapi_sender(), the application callback function may receive RAPI_RESV_EVENT or RAPI_PATH_ERROR events.

Return Values and Descriptions

RAPI_ERR_BADSID	The session identifier is not valid, or the session is not open.
RAPI_ERR_INVAL	An argument is not valid.
RAPI_ERR_NORSVP	The RSVP daemon in not running.
RAPI_ERR_NOTSPEC	No traffic spec (tspec) was specified.
RAPI_ERR_OK	The call succeeded.

Related Calls

rapi_reserve()

----

rapi_session()

The rapi_session() call starts an RSVP session.

Syntax

```
       #include <rsvprapi.h>
       #include <netinet\in.h>

rapi_sid_t  rapi_session(
            struct sockaddr *pdest,
            int              protoId,
            int              flags,
            rapi_event_rtn_t caller,
            void            *pClientArg,
            int             *perror);
```

Parameters

pdest

A pointer to a sockaddr structure that defines an address and port for the destination. The sockaddr structure should be sockaddr_in. For multicast, this should be the multicast group and port. Otherwise, it should be the address and port of the receiver of the unicast data stream.

protoId

Protocol to be used for the data stream. For example, IPPROTO_UDP can be used for UDP (unicast or multicast), or IPPROTO_TCP can be used for TCP (unicast). If *protoId* is 0, it is set by default to IPPROTO_UPD.

flags

No flags are defined at this time; specify 0.

caller

A pointer to a callback function to be used for asynchronous events. The pointer may be NULL, indicating that there is no such routine.

pClientArg

A user supplied parameter that will be passed to the callback function. The parameter may be NULL.

perror

A pointer to a variable in which error codes are passed.

Description

The rapi_session() call creates an RSVP API session. The session ID is an opaque non-zero value that refers to the session until it is released with rapi_release(). It is not useful to compare session IDs from different processes or from different hosts on the network.

After rapi_session() is successfully called, the application callback function may receive RAPI_PATH_EVENT messages.

Return Values and Descriptions

A non-zero return value is a new session ID which is used as a handle in subsequent calls to the API. A zero return value (NULL_SID) indicates an error, in which case an error code is in the error code variable pointed to by the perror parameter.

If the return code is zero, rapi_session() stores these values in the error code variable:

RAPI_ERR_NORSVP	The RSVP daemon in not running.
RAPI_ERR_SYSCALL	System call error; see errno.
RAPI_ERR_MAXSESS	Too many sessions.
RAPI_ERR_OK	No error.

Related Calls

rapi_release()
user_rapi_callback()

----

rapi_version()

The rapi_version() call gets the RSVP version.

Syntax

```
#include <rsvprapi.h>

int rapi_version(void);
```

Return Values

The rapi_version() call returns the version of the RSVP API. The integer value is encoded as major*100 + minor. The major number of this release is 4.

----

user_rapi_callback()

The user_rapi_callback() call provides the RSVP callback function.

Syntax

```
       #include <rsvprapi.h>

int  _System user_rapi_callback(
       rapi_sid_t         sid,
       rapi_eventinfo_t   eventType,
       rapi_styleid_t     styleID,
       int                errorCode,
       int                errorValue,
       struct sockaddr   *pErrorNodeAddr,
       u_char             errorFlags,
       int                nFilterSpecs,
       rapi_filter_t     *pFilterSpec,
       int                nFlowSpecs,
       rapi_flowspec_t   *pFlowSpec,
       int                nAdSpecs,
       rapi_adspec_t     *pAdSpec,
       void              *pClientArg);
```

Parameters

sid

Session identifier of the session that generated the event.

eventType

One of these event types:

RAPI_PATH_EVENT	A path event is generated by a rapi_sender() call for the session. This can be received by a program that has called rapi_session(). A non-zero value of nFlowSpecs indicates that a path exists from a sender to a potential receiver. A zero value of nFlowSpecs indicates that a previous path may have gone away.
RAPI_RESV_EVENT	A reservation event is generated by a rapi_reserve() call for the session. This is received by a sender to which the reservation applies. A zero value of nFlowSpecs indicates that a previous reservation may have gone away.
RAPI_PATH_ERROR	There has been an error associated with the path event. See errorCode for the type of error.
RAPI_RESV_ERROR	There has been an error associated with the reservation event. See errorCode for the type of error.
RAPI_RESV_CONFIRM	When a rapi_reserve() call specifies that confirmation of the reservation is requested, this event can be generated to confirm (with a very high probability) that the reservation has been made from receiver to sender.

styleID

Reservation style. This is non-zero only for RAPI_RESV_EVENT and RAPI_RESV_ERROR.

errorCode

Type of error that occurred, for a RAPI_PATH_ERROR or RAPI_RESV_ERROR event.

errorValue

Extra error value for a RAPI_PATH_ERROR or RAPI_RESV_ERROR event. This parameter is used if it is necessary for RSVP to report an extra error. An example might be an "errno" value.

pErrorNodeAddr

A pointer to the IP address and port of the node that detected an error. For this implementation, this is a pointer to a sockaddr_in structure. This is set only for a RAPI_PATH_ERROR or RAPI_RESV_ERROR event.

errorFlags

Error flag for a RAPI_PATH_ERROR event. Set to one of these values:

RAPI_ERRF_InPlace	This value indicates that the reservation failed, but another probably smaller reservation was left in place at the failing node address.

RAPI_ERRF_NotGuilty	This value indicates that the flowspec that was requested by this receiver was not the cause of the error, even though the reservation failed. Presumably the failure was due to a larger reservation that this one was merged with.

nFilterSpecs
The number of filterspecs or sender templates pointed to by pFilterSpec.

pFilterSpec
A pointer to an array of filterspecs or sender templates, or NULL.

nFlowSpecs
The number of flowspecs or tspecs pointed to by pFlowSpec.

pFlowSpec
A pointer to an array of flowspecs or tspecs, or NULL.

nAdSpecs
The number of adspecs pointed to by pAdSpec.

pAdSpec
A pointer to an array of adspecs, or NULL.

pClientArg
The client-supplied argument that was provided when rapi_session() was called.

Description

The user_rapi_callback() call processes asynchronous events from the RSVP API. A pointer to the function is passed to rapi_session(). The callback function must be declared as shown above. The name of the function is chosen by the user, but is shown above as user_rapi_callback().

The rapi_getfd() call returns a socket for the session. The socket may be used with the select() call as a read socket. When a select() call indicates that there is some data to read, a rapi_dispatch() call should be issued to read and process the data from the socket. Typically the rapi_dispatch() call will invoke the user callback function to provide the program with information, if the user supplied such a function in the rapi_session() call.

The user callback function should copy any information that it wants to save, because the storage that is pointed to by the pErrorNodeAddr, pFilterSpec, pFlowSpec, and pAdSpec will be freed as soon as the callback function returns.

Return Values

The function is declared to have an integer return value, but nothing is done with the value currently. It is recommended that the user always return 0.

Related Calls

rapi_dispatch()
rapi_getfd()
rapi_session()
select()

Appendixes

This section describes:

- NETWORKS File Structure

  Provides examples of network names contained in the TCPIP\ETC\NETWORKS file.

- Socket Error Constants

  Provides the socket error codes and descriptions.

- Well-Known Port Assignments

Provides a list of the well-known ports supported by TCP/IP.

- Notices

  Contains copyright notices, disclaimers, and trademarks relating to TCP/IP for OS/2 Warp.

NETWORKS File Structure

The NETWORKS file contains the network name, number, and alias or aliases of known networks. The NETWORKS file must reside in the directory specified by the ETC environment variable. The NETWORKS file is used only by the following socket calls:

- endnetent()
- getnetbyaddr()
- getnetbyname()
- getnetent()
- setnetent()

The following table lists examples of network names contained in the NETWORKS file.

Name Structures of Known Networks

Name of File	Contents of File	Sample File Entries
NETWORKS	*official_network_name network_number alias*	ne-region 128.1 classb.net1 at1-region 128.2 classb.net2 lab-net 192.5.1 classc.net5

Socket Error Constants

The following table provides the error constants set by socket calls. This table can be found in the <NERRNO.H> header file.

```
/*
* The redefinition of error constants is necessary to avoid conflict with
* standard compiler error constants.
*
* All OS/2 SOCKETS API error constants are biased by SOCBASEERR from the "normal"
*
*/

#define SOCBASEERR              10000

/*
* OS/2 SOCKETS  API definitions of regular Microsoft C 6.0 error constants
*/

#define SOCEPERM            (SOCBASEERR+1)   /*Not owner*/
#define SOCESRCH            (SOCBASEERR+3)   /*No such process*/
#define SOCEINTR            (SOCBASEERR+4)   /*Interrupted system call*/
#define SOCENXIO            (SOCBASEERR+6)   /*No such device or address*/
#define SOCEBADF            (SOCBASEERR+9)   /*Bad file number*/
#define SOCEACCES           (SOCBASEERR+13)  /*Permission denied*/
#define SOCEFAULT           (SOCBASEERR+14)  /*Bad address*/
#define SOCEINVAL           (SOCBASEERR+22)  /*Invalid argument*/
#define SOCEMFILE           (SOCBASEERR+24)  /*Too many open files*/
#define SOCEPIPE            (SOCBASEERR+32)  /*Broken pipe*/

#define SOCEOS2ERR          (SOCBASEERR+100) /*OS/2 Error*/
```

```
/*
* OS/2 SOCKETS API definitions of regular BSD error constants
*/

#define SOCEWOULDBLOCK     (SOCBASEERR+35)  /*Operation would block*/
#define SOCEINPROGRESS     (SOCBASEERR+36)  /*Operation now in progress*/
#define SOCEALREADY        (SOCBASEERR+37)  /*Operation already in progress*/
#define SOCENOTSOCK        (SOCBASEERR+38)  /*Socket operation on non-socket*/
#define SOCEDESTADDRREQ    (SOCBASEERR+39)  /*Destination address required*/
#define SOCEMSGSIZE        (SOCBASEERR+40)  /*Message too long*/
#define SOCEPROTOTYPE      (SOCBASEERR+41)  /*Protocol wrong type for socket*/
#define SOCENOPROTOOPT     (SOCBASEERR+42)  /*Protocol not available*/
#define SOCEPROTONOSUPPORT (SOCBASEERR+43)  /*Protocol not supported*/
#define SOCESOCKTNOSUPPORT (SOCBASEERR+44)  /*Socket type not supported*/
#define SOCEOPNOTSUPP      (SOCBASEERR+45)  /*Operation not supported on socket*/
#define SOCEPFNOSUPPORT    (SOCBASEERR+46)  /*Protocol family not supported*/
#define SOCEAFNOSUPPORT    (SOCBASEERR+47)  /*Address family not supported by protocol family*/
#define SOCEADDRINUSE      (SOCBASEERR+48)  /*Address already in use*/
#define SOCEADDRNOTAVAIL   (SOCBASEERR+49)  /*Can't assign requested address*/
#define SOCENETDOWN        (SOCBASEERR+50)  /*Network is down*/
#define SOCENETUNREACH     (SOCBASEERR+51)  /*Network is unreachable*/
#define SOCENETRESET       (SOCBASEERR+52)  /*Network dropped connection on reset*/
#define SOCECONNABORTED    (SOCBASEERR+53)  /*Software caused connection abort*/
#define SOCECONNRESET      (SOCBASEERR+54)  /*Connection reset by peer*/
#define SOCENOBUFS         (SOCBASEERR+55)  /*No buffer space available*/
#define SOCEISCONN         (SOCBASEERR+56)  /*Socket is already connected*/
#define SOCENOTCONN        (SOCBASEERR+57)  /*Socket is not connected*/
#define SOCESHUTDOWN       (SOCBASEERR+58)  /*Can't send after socket shutdown*/
#define SOCETOOMANYREFS    (SOCBASEERR+59)  /*Too many references: can't splice*/
#define SOCETIMEDOUT       (SOCBASEERR+60)  /*Connection timed out*/
#define SOCECONNREFUSED    (SOCBASEERR+61)  /*Connection refused*/
#define SOCELOOP           (SOCBASEERR+62)  /*Too many levels of symbolic links*/
#define SOCENAMETOOLONG    (SOCBASEERR+63)  /*File name too long*/
#define SOCEHOSTDOWN       (SOCBASEERR+64)  /*Host is down*/
#define SOCEHOSTUNREACH    (SOCBASEERR+65)  /*No route to host*/
#define SOCENOTEMPTY       (SOCBASEERR+66)  /*Directory not empty*/

/*
* OS/2 SOCKETS API errors redefined as regular BSD error constants
*/

#define EWOULDBLOCK         SOCEWOULDBLOCK
#define EINPROGRESS         SOCEINPROGRESS
#define EALREADY            SOCEALREADY
#define ENOTSOCK            SOCENOTSOCK
#define EDESTADDRREQ        SOCEDESTADDRREQ
#define EMSGSIZE            SOCEMSGSIZE
#define EPROTOTYPE          SOCEPROTOTYPE
#define ENOPROTOOPT         SOCENOPROTOOPT
#define EPROTONOSUPPORT     SOCEPROTONOSUPPORT
#define ESOCKTNOSUPPORT     SOCESOCKTNOSUPPORT
#define EOPNOTSUPP          SOCEOPNOTSUPP
#define EPFNOSUPPORT        SOCEPFNOSUPPORT
#define EAFNOSUPPORT        SOCEAFNOSUPPORT
#define EADDRINUSE          SOCEADDRINUSE
#define EADDRNOTAVAIL       SOCEADDRNOTAVAIL
#define ENETDOWN            SOCENETDOWN
#define ENETUNREACH         SOCENETUNREACH
#define ENETRESET           SOCENETRESET
#define ECONNABORTED        SOCECONNABORTED
#define ECONNRESET          SOCECONNRESET
#define ENOBUFS             SOCENOBUFS
#define EISCONN             SOCEISCONN
#define ENOTCONN            SOCENOTCONN
#define ESHUTDOWN           SOCESHUTDOWN
#define ETOOMANYREFS        SOCETOOMANYREFS
#define ETIMEDOUT           SOCETIMEDOUT
#define ECONNREFUSED        SOCECONNREFUSED
#define ELOOP               SOCELOOP
#define ENAMETOOLONG        SOCENAMETOOLONG
#define EHOSTDOWN           SOCEHOSTDOWN
#define EHOSTUNREACH        SOCEHOSTUNREACH
#define ENOTEMPTY           SOCENOTEMPTY
```

Well-Known Port Assignments

The following table is a list of the common well-known ports supported by TCP/IP. It provides the port number, keyword, and a description of the reserved port assignment. Port numbers of less than 1024 are reserved for system applications. You can also find a complete list of well-known port numbers in the ETC\SERVICES file.

TCP Well-Known Port Assignments

TCP Well-Known Port Assignments

Port Number	Keyword	Reserved for	Services Description
0		reserved	
5	**rje**	remote job entry	remote job entry
7	**echo**	echo	echo
9	**discard**	discard	sink null
11	**systat**	active users	active users
13	**daytime**	daytime	daytime
15	**netstat**	Netstat	who is up or Netstat
19	**chargen**	ttytst source	character generator
21	**ftp**	FTP	File Transfer Protocol
23	**telnet**	Telnet	Telnet
25	**smtp**	mail	Simple Mail Transfer Protocol
37	**time**	timeserver	timeserver
39	**rlp**	resource	Resource Location Protocol
42	**nameserver**	name	host name server
43	**nicname**	who is	who is
53	**domain**	name server	domain name server
57	**mtp**	private terminal access	private terminal access
67	**bootps**	bootps dhcps	bootp server
68	**bootpc**	bootpc dhcpc	bootp client
69	**tftp**	TFTP	Trivial File Transfer Protocol
70	**gopher**	gopher	Gopher
77		netrjs	any private RJE service
79	**finger**	finger	finger

80	**www-http**	www-http	World Wide Web HTTP
87	**link**	ttylink	any private terminal link
95	**supdup**	supdup	SUPDUP Protocol
101	**hostname**	hostname	nic hostname server, usually from SRI-NIC
109	**pop**	postoffice	Post Office Protocol
111	**sunrpc**	sunrpc	Sun remote procedure call
113	**auth**	authentication	authentication service
115	**sftp**	sftp	Simple File Transfer Protocol
117	**uucp-path**	UUCP path service	UUCP path service
119	**untp**	readnews untp	USENET News Transfer Protocol
123	**ntp**	NTP	Network Time Protocol
160		reserved	
163-223		reserved	
449	AS-SVRMAP	mapper function for AS/400 servers	servers for sign-on, central management, network print, database, stream file, data queue, and remote command and distributed program calls.
712	**vexec**	vice-exec	Andrew File System authenticated service
713	**vlogin**	vice-login	Andrew File System authenticated service
714	**vshell**	vice-shell	Andrew File System authenticated service
2001	**filesrv**		Andrew File System service
2106	**venus.itc**		Andrew File System service, for the Venus process

UDP Well-Known Port Assignments

UDP Well-Known Port Assignments

Port Number	Keyword	Reserved for	Services Description
0		reserved	
5	**rje**	remote job entry	remote job entry
7	**echo**	echo	echo
9	**discard**	discard	sink null
11	**users**	active users	active users
13	**daytime**	daytime	daytime
15	**netstat**	Netstat	Netstat
19	**chargen**	ttytst source	character generator
37	**time**	timeserver	timeserver
39	**rlp**	resource	Resource Location Protocol
42	**nameserver**	name	host name server
43	**nicname**	who is	who is
53	**domain**	name server	domain name server
67	**bootps**	bootps dhcps	bootp server
68	**bootpc**	bootpc dhcpc	bootp client
69	**tftp**	TFTP	Trivial File Transfer Protocol
70	**gopher**	gopher	Gopher
75			any private dial out service
77		netrjs	any private RJE service
79	**finger**	finger	finger
80	**www-http**	www-http	World Wide Web HTTP
111	**sunrpc**	sunrpc	Sun remote procedure call
123	**ntp**	NTP	Network Time Protocol
135	**llbd**	NCS LLBD	NCS local location broker daemon
160-223		reserved	
531	**rvd-control**		rvd control port
2001	**rauth2**		Andrew File System service, for the Venus process

2002	**rfilebulk**	Andrew File System service, for the Venus process
2003	**rfilesrv**	Andrew File System service, for the Venus process
2018	**console**	Andrew File System service
2115	**ropcons**	Andrew File System service, for the Venus process
2131	**rupdsrv**	assigned in pairs; bulk must be **srv** +1
2132	**rupdbulk**	assigned in pairs; bulk must be **srv** +1
2133	**rupdsrv1**	assigned in pairs; bulk must be **srv** +1
2134	**rupdbulk1**	assigned in pairs; bulk must be **srv** +1

Notices

Fourth Edition (February 1999)

Copyright Notices

Disclaimers

Acknowledgments

TCP/IP for OS/2 incorporates compression code by the Info-ZIP group. There are no extra charges or costs due to the use of this code, and the original compression sources are freely available from Compuserve in the OS2USER forum and by anonymous ftp from the Internet site ftp.uu.net:/pub/archiving/zip.

Trademarks
